目錄

Introduction 1.1
写在前面 1.2
关于本书 1.3
这本书适合你吗？ 1.4
关于作者 1.5
介绍 1.6
- 什么是Kotlin？ 1.6.1
- 我们通过Kotlin得到什么 1.6.2

准备工作 1.7
- Android Studio 1.7.1
- 安装Kotlin插件 1.7.2

创建一个新的项目 1.8
- 在Android Studio中创建一个项目 1.8.1
- 配置Gradle 1.8.2
- 把MainActivity转换成Kotlin代码 1.8.3
- 测试是否一切就绪 1.8.4

类和函数 1.9
- 怎么定义一个类 1.9.1
- 类继承 1.9.2
- 函数 1.9.3
- 构造方法和函数参数 1.9.4

编写你的第一个类 1.10
- 创建一个layout 1.10.1
- The Recycler Adapter 1.10.2

变量和属性 1.11
- 基本类型 1.11.1
- 变量 1.11.2

属性 1.11.3
Anko和扩展的函数 1.12
Anko是什么？ 1.12.1
开始使用Anko 1.12.2
扩展函数 1.12.3
从API中获取数据 1.13
执行一个请求 1.13.1
在主线程以外执行请求 1.13.2
数据类 1.14
额外的函数 1.14.1
复制一个数据类 1.14.2
映射对象到变量中 1.14.3
解析数据 1.15
转换json到数据类 1.15.1
构建domain层 1.15.2
在UI中绘制数据 1.15.3
操作符重载 1.16
操作符表 1.16.1
例子 1.16.2
扩展函数中的操作符 1.16.3
使Forecast list可点击 1.17
Lambdas 1.18
简化setOnClickListener() 1.18.1
ForecastListAdapter的click listener 1.18.2
扩展语言 1.18.3
可见性修饰符 1.19
修饰符 1.19.1
构造器 1.19.2
润色我们的代码 1.19.3
Kotlin Android Extensions 1.20

- 怎么去使用Kotlin Android Extensions 1.20.1
- 重构我们的代码 1.20.2

Application单例化和属性的Delegated 1.21

- Applicaton单例化 1.21.1
- 委托属性 1.21.2
- 标准委托 1.21.3
- 怎么去创建一个自定义的委托 1.21.4
- 重新实现Application单例化 1.21.5

创建一个SQLiteOpenHelper 1.22

- ManagedSqliteOpenHelper 1.22.1
- 定义表 1.22.2
- 实现SqliteOpenHelper 1.22.3
- 依赖注入 1.22.4

集合和函数操作符 1.23

- 总数操作符 1.23.1
- 过滤操作符 1.23.2
- 映射操作符 1.23.3
- 元素操作符 1.23.4
- 生产操作符 1.23.5
- 顺序操作符 1.23.6

从数据库中保存或查询数据 1.24

- 创建数据库model类 1.24.1
- 写入和查询数据库 1.24.2

Kotlin中的null安全 1.25

- 可null类型怎么工作 1.25.1
- 可null性和Java库 1.25.2

创建业务逻辑来访问数据 1.26

Flow control和ranges 1.27

- If表达式 1.27.1
- When表达式 1.27.2

For循环 1.27.3
While和do/while循环 1.27.4
Ranges 1.27.5
创建一个详情界面 1.28
准备请求 1.28.1
提供一个新的activity 1.28.2
启动一个activity：reified函数 1.28.3
接口和委托 1.29
接口 1.29.1
委托 1.29.2
在我们的App中实现一个例子 1.29.3
泛型 1.30
基础 1.30.1
变体 1.30.2
泛型例子 1.30.3
设置界面 1.31
创建一个设置activity 1.31.1
访问Shared Preferences 1.31.2
泛型preference委托 1.31.3
测试你的App 1.32
Unit testing 1.32.1
Instrumentation tests 1.32.2
其它的概念 1.33
内部类 1.33.1
枚举 1.33.2
密封（Sealed）类 1.33.3
异常（Exceptions） 1.33.4
结尾 1.34

《Kotlin for android developers》中文版翻译

> 错别字、病句、翻译错误等问题可以提issues。请说明错误原因。

1. 在线阅读或下载GitBook
2. 在线阅读

希望大家购买正版，建议阅读英文原版：https://leanpub.com/kotlin-for-android-developers

- Github: https://github.com/wangjiegulu/kotlin-for-android-developers-zh

最后打个广告哈，阿里巴巴，杭州招技术，支持电话视频面试，有兴趣的同学戳这里：https://github.com/wangjiegulu/jobs

写在前面

学习通过Kotlin语言来简单地开发android应用。

关于本书

在这本书中，我会使用Kotlin作为主要的语言来开发一个android应用。方式是通过开发一个应用来学习这门语言，而不是根据传统的结构来学习。我会在感兴趣的点停下来通过与Java1.7对比的方式讲讲Kotlin的一些概念和特性。用这种方法你就能知道它们的不同之处，并且知道哪部分语言特性可以让你提高你的工作效率。

这本书并不是一本语言参考书，但它是一个Android开发者去学习Kotlin并且使用在自己项目中的一个工具。我会通过使用一些语言特性和有趣的工具和库来解决很多我们在日常生活当中都会遇到的典型问题。

这本书是非常具有实践性的，所以我建议你在电脑面前跟着我的例子和代码实践。无论何时你都可以在有一些想法的时候深入到实践中去。

就如你知道的，这是一个精益出版。也就是说这本书是跟你一起写下去的。我会根据你的回复和建议来写新的内容和检查之前的内容。尽管这本书已经完成了，但是我会及时根据新的Kotlin版本更新。所以尽管编写意见告诉我你对这本书的看法，或者需要改进的地方。我希望这本书会成为Android开发者的一个完美的工具，正因为如此，欢迎大家的想法和帮助。

感谢你将成为这个激动人心的项目的一部分。

这本书适合你吗？

写这本书是为了帮助那些有兴趣使用Kotlin语言来进行开发的Android开发者。

如果你符合下面这些情况，那这本书是适合你的：

- 你有相关Android开发和Android SDK的基本知识。
- 你希望跟随一个使用Kotlin语言编写的例子来学习Kotlin。
- 你需要一个怎么去使用更简洁生动的语言来解决日常生活遇到的典型问题的指南。

另一方面，这本书可能不太适合你，因为：

- 这本书不是Kotlin圣经。我会去解释所有Kotlin的基本语法，甚至包括在过程中遇到我需要的一些相对比较复杂的想法。所以你是通过一个例子去学习，而不是其他方式。
- 我不会去解释怎么样去开发一个Android应用。你不需要很深的开发知识，但是至少了解基础，比如Android Studio，Gradle，Java语言和Android SDK。你可能会从中学到一些关于Android开发的一些新的东西。
- 这本书不是函数式编程语言指南。当然由于Java 7完全不是函数式风格的，我会解释你需要知道的东西，但是不会很深入地去讲解函数式编程的话题。

关于作者

Antonio Leiva是一个Android工程师，他专注于研究新的潜在的Android开发可能性，然后写作说明。他维护一个关于很多不同Android开发话题的博客antonioleiva.com。

Antonio一开始是CRM技术顾问，但是一段时间之后，他寻找着新的激情，他发现了Android世界。在优秀的平台上获得了相关经验，之后他加入了一个西班牙重要的手机公司带领多个项目作为新的冒险。

现在，他在Plex担任Android工程师，并且在Android的设计和UX方面也担任重要的角色。

你可以在Twitter上关注他@lime_cl。

介绍

如果你觉得Java 7是一个过期的语言，并决定找一个更现代的语言代替。恭喜你！就如你知道的，虽然Java 8已经发布了，它包含了很多我们期待的像现代语言中那样的改善，但是我们Android开发者还是被迫在使用Java 7.这是因为法律的问题。但是就算没有这个限制，并且新的Android设备从今天开始使用新的能理解Java8的VM，在当前的设备过期、几乎没有人使用它们之前我们也不能使用Java 8，所以恐怕我们不会很快等到这一天的到来。

但是并不是没有补救的方法。多亏使用了JVM，我们可以使用任何语言去编写Android应用，只要它能够编译成JVM能够认识的字节码就可以了。

正如你所想，有很多选择，比如Groovy，Scala，Clojure，当然还有Kotlin。通过实践，只有其中一些能够被考虑来作为替代品。

上述的每一种语言都有它的利弊，如果你还没有眞正确定你该使用那种语言，我建议你可以去尝试一下它们。

什么是Kotlin？

Kotlin，如前面所说，它是JetBrains开发的基于JVM的语言。JetBrains因为创造了一个强大的Java开发IDE被大家所熟知。Android Studio，官方的Android IDE，就是基于Intellij，作为一个该平台的插件。

Kotlin是使用Java开发者的思维被创建的，Intellij作为它主要的开发IDE。对于Android开发者，有两个有趣的特点：

- 对Java开发者来说，Kotlin是非常直觉化的，并且非常容易学习。语言的大部分内容都是与我们知道的非常相似，不同的地方，它的基础概念也能迅速地掌握它。
- 它与我们日常生活使用的IDE无需配置就能完全整合。Android Studio能够非常完美地理解、编译运行Kotlin代码。而且对这门语言的支持正是来自于开发了这个IDE的公司本身，所以我们Android开发者是一等公民。

但是这仅仅是开发语言和开发工具之间的整合。相比Java 7的优势到底是什么呢？

- 它更加易表现：这是它最重要的优点之一。你可以编写少得多的代码。
- 它更加安全：Kotlin是空安全的，也就是说在我们编译时期就处理了各种null的情况，避免了执行时异常。如果一个对象可以是null，则我们需要明确地指定它，然后在使用它之前检查它是否是null。你可以节约很多调试空指针异常的时间，解决掉null引发的bug。
- 它是函数式的：Kotlin是基于面向对象的语言。但是就如其他很多现代的语言那样，它使用了很多函数式编程的概念，比如，使用lambda表达式来更方便地解决问题。其中一个很棒的特性就是Collections的处理方式。
- 它可以扩展函数：这意味着我们可以扩展类的更多的特性，甚至我们没有权限去访问这个类中的代码。
- 它是高度互操作性的：你可以继续使用所有的你用Java写的代码和库，因为两个语言之间的互操作性是完美的。甚至可以在一个项目中使用Kotlin和Java两种语言混合编程。

我们通过Kotlin得到什么

不深入Kotlin语言（我们会在下一章再去学习），这里有一些Java中没有的有趣的特性：

易表现

通过Kotlin，可以更容易地避免模版代码因为大部分的典型情况都在语言中默认覆盖实现了。举个例子，在Java中，如果我们要典型的数据类，我们需要去编写（至少生成）这些代码：

```
public class Artist {
    private long id;
    private String name;
    private String url;
    private String mbid;

    public long getId() {
        return id;
    }


    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getUrl() {
        return url;
    }
```

```
    public void setUrl(String url) {
        this.url = url;
    }

    public String getMbid() {
        return mbid;
    }

    public void setMbid(String mbid) {
        this.mbid = mbid;
    }

    @Override public String toString() {
        return "Artist{" +
          "id=" + id +
          ", name='" + name + '\'' +
          ", url='" + url + '\'' +
          ", mbid='" + mbid + '\'' +
          '}';
    }
}
```

使用Kotlin，我们只需要通过数据类：

```
data class Artist(
    var id: Long,
    var name: String,
    var url: String,
    var mbid: String)
```

这个数据类，它会自动生成所有属性和它们的访问器，以及一些有用的方法，比如， `toString()`

空安全

当我们使用Java开发的时候，我们的代码大多是防御性的。如果我们不想遇到 `NullPointerException` ，我们就需要在使用它之前不停地去判断它是否为null。Kotlin，如很多现代的语言，是空安全的，因为我们需要通过一个 `安全调用操作符` （写做 `?` ）来明确地指定一个对象是否能为空。

我们可以像这样去写：

```
// 这里不能通过编译. Artist 不能是null
var notNullArtist: Artist = null

// Artist 可以是 null
var artist: Artist? = null

// 无法编译, artist可能是null，我们需要进行处理
artist.print()

// 只要在artist != null时才会打印
artist?.print()

// 智能转换. 如果我们在之前进行了空检查，则不需要使用安全调用操作符调用
if (artist != null) {
  artist.print()
}

// 只有在确保artist不是null的情况下才能这么调用，否则它会抛出异常
artist!!.print()

// 使用Elvis操作符来给定一个在是null的情况下的替代值
val name = artist?.name ?: "empty"
```

扩展方法

我们可以给任何类添加函数。它比那些我们项目中典型的工具类更加具有可读性。举个例子，我们可以给fragment增加一个显示toast的函数：

```
fun Fragment.toast(message: CharSequence, duration: Int = Toast.
LENGTH_SHORT) {
    Toast.makeText(getActivity(), message, duration).show()
}
```

我们现在可以这么做：

```
fragment.toast("Hello world!")
```

函数式支持（Lambdas）

每次我们去声明一个点击所触发的事件，可以只需要定义我们需要做些什么，而不是不得不去实现一个内部类？我们确实可以这么做，这个（或者其它更多我们感兴趣的事件）我们需要感谢lambda：

```
view.setOnClickListener { toast("Hello world!") }
```

这里只是挑选了很小一部分Kotlin可以简化我们代码的事情。现在你已经知道这门语言的一些有趣的特性了，你可以考虑它是否是适合你的。如果你选择继续，我们将在下一章开始我们的实践之旅。

准备工作

现在你知道使用Kotlin实现的小例子了，我确信你会希望尽可能快地把它用在你的实践当中去。不要担心，在第一章中会帮助你去搭建你的开发环境，这样你才能立即编写代码。

Android Studio

第一件事就是安装Android Studio。就如你知道的，Android Studio是官方的Android IDE，它是2013年发布的预览版，并在2014年发布了正式版。

Android Studio是Intellij IDEA的插件实现，Intellij IDEA是由JetBrains开发，Kotlin就是JetBrains创造的。所以，正如你所见，一切都这么紧密地结合起来了。

转移到Android Studio是Android开发者一个重要的改变。首先，因为我们放弃了Eclipse并转到专为Java开发者设计的完美的语言交互的软件。我们可以享受到完美的特性体验，比如反应快速和令人影响深刻的智能代码提示，还有强大的分析和重构工具。

第二，Gradle成为Android官方的系统构建工具，这意味着版本构建和部署的新的可能性。最有趣的两点是系统构建和flavours，它可以让你使用相同的代码库来创建无限的版本（甚至是不同的应用）。

如果你仍然在使用Eclipse，为了跟上这本书，恐怕你需要转移到Android Studio了。Kotlin团队也创建了一个针对Eclipse的插件，但是它是远远落后于Android Studio的，而且结合得也并不完美。你一旦使用了它，你就会觉得相见恨晚。

我不会去覆盖到Android Studio和Gradle的使用，因为这些都不是本书的重点，但是如果你以前没有使用过这些工具，不要恐慌，我确信你能够跟随本书的同时学习到相关基础。

如果你还没有AndroidStudio，点这里从官网下载。

安装Kotlin插件

~~IDE它本身并不能理解Kotlin。就像前面部分讲到，它是为Java开发设计的。但是Kotlin团队创建了一系列强大的插件让我们更轻松地实现。前往Android Studio的 `Preferences` 中 `Plugin` 栏，然后安装如下两个插件：~~

- ~~Kotlin：这是一个基础的插件。它能让Android Studio懂得Kotlin代码。它会每次在新的Kotlin语言版本发布的时候发布新的插件版本，这样我们可以通过它发现新版本特性和弃用的警告。这是你要使用Kotlin编写Android应用唯一的插件。但是我们现在还需要另外一个。~~
- ~~Kotlin Android Extensions：Kotlin团队还为Android开发发布了另外一个有趣的插件。这个Android Extensions可以让你自动地从XML中注入所有的View到Activity中，举个例子，你不需要使用 `findViewById()` 。你将会立即得到一个从属性转换过来的view。你将需要安装这个插件来使用这个特性。我们会在下一章中深入地去讲解这个。~~

因为从Intellij 15开始，插件是被默认安装了的，但是你的Android Studio可能并没有。所以你需要进入Android Studio 的Preferences的plugin栏，然后安装Kotlin插件。如果你不会就去搜索下。

现在我们的环境已经可以理解Kotlin语言了，可以就像我们使用Java一样无缝地编译它，执行它。

创建一个新的项目

如果你已经使用过Android Studio和Gradle，那么这一章会比较简单。我不会给出很多细节和截图，因为用户界面和细节可能会一直变化。

我们的应用是由一个简单的天气app组成，正如所使用的Google's Beginners Course in Udacity。我们可能会关注不同的事情，但是app的想法都是一样的，你会发现在一个典型的app里面会包括很多不同的东西。如果你的Android开发水平比较低，我推荐这个，这个过程是比较容易的。

在Android Studio中创建一个项目

首先，打开Android Studio并选择 `Create new Project` ，然后它会让你输入一个名字，你可以任意取一个名字，比如： `Weather App` 。然后你需要输入公司域名。如果你不会真正发布这个app，这个字段就不是特别重要了，但是如果你有的话可以使用自己的域名。然后任意选择一个目录作为这个项目的保存地址。

下一步，它会让你选择最小的API版本。我们选择API 15，因为我们有一个库需要至少API 15才能用。无论如何你把大部分的Anroid用户作为了目标。现在不要选择任何除了手机和平板的其它平台。

最后，我们需要选择一个Activity模版来作为入口。我们可以选择 `Add no Activity` 然后从头开始（这是一个好的方式如果这是一个Kotlin项目的话），但是我将选择 `Blank Activity` ，因为我待会儿会给你展示Kotlin插件一个好玩的小特性。

暂时不用去关心Activity的名字，layout等。这些你会在下一篇中知道。如果我们需要，我待会儿会修改它。点击 `Finish` 然后让它继续创建项目。

配置Gradle

Kotlin插件包括一个让我们配置Gradle的工具。但是我还是倾向于保持我对Gradle文件读写的控制权，否则它只会变得混乱而不会变得简单。不管怎么样，在使用自动工具之前知道它是怎么工作的是个不错的主意。所以这次，我们将手动去做。

首先，你需要如下修改父 `build.gradle` ：

```
buildscript {
    ext.support_version = '23.1.1'
    ext.kotlin_version = '1.0.0'
    ext.anko_version = '0.8.2'
    repositories {
        jcenter()
        dependencies {
            classpath 'com.android.tools.build:gradle:1.5.0'
            classpath "org.jetbrains.kotlin:kotlin-gradle-plugin
:$kotlin_version"
        }
    }
}
allprojects {
    repositories {
        jcenter()
    }
}
```

正如你看到的，我们创建了一个变量来存储当前的Kotlin版本。你读到这里的时候去检测一下最新版本，因为可能会有更新的版本已经发布了。我们需要在几个不同的地方用到那个版本号，比如你需要加上新的Kotlin插件的 `dependency` 。你会在你指定的那些模块中的 `build.gradle` 中再次需要到Kotlin标准库。

我们对于 `support library` 也是如此， `Anko` 库也是同样的做法。用这个方式可以更方便地在一个地方修改所有的版本号。并且使用相同的版本号，更新的时候也不需要每个地方都修改。

我们会增加 `Kotlin` 标准库， `Anko` 库，以及 `Kotlin` 和 `Kotlin Android Extensions plugin` 插件到dependencies。

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
android {
    ...
}

dependencies {
    compile "com.android.support:appcompat-v7:$support_version"
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    compile "org.jetbrains.anko:anko-common:$anko_version"
}

buildscript {
    repositories {
      jcenter()
    }
    dependencies {
      classpath "org.jetbrains.kotlin:kotlin-android-extensions:
$kotlin_version"
    }
}
```

Anko是一个用来简化一些Android任务的很强大的Kotlin库。我们之后将会学习部分anko，但是现在来说仅仅增加 `anko-common` 就足够了。这个库被分割成了一系列小的部分以至于我们不会把没用到的部分加进来。

把MainActivity转换成Kotlin代码

Kotlin plugin包含了一个有趣的特性，它能把Java代码转成Kotlin代码。正如任何自动化那样，结果不会很完美，但是在你第一天能够使用Kotlin语言开始编写代码之前，它还是提供了很多的帮助。

所以我们在MainActivity.java类中使用它。打开文件，然后选择`Code` -> `Convert Java File to Kotlin File`。对比它们的不同之处，可以让你更熟悉这门语言。

测试是否一切就绪

我们想再将编写一些代码来测试Kotlin Android Extensions是否在工作。我现在还不会对这些代码做解释，但是我想要确保它们在你的环境中是正常运行的。这可能是配置中最难的一部分。

首先，打开 `activity_main.xml` ，然后设置TextView的id：

```
<TextView
    android:id="@+id/message"
    android:text="@string/hello_world"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

然后，手动在Activity中增加一个import语句（不要担心你现在对这个还不太理解）。

```
import kotlinx.android.synthetic.main.activity_main.*
```

在 `onCreate` 中，你现在可以直接得到并访问这个TextView了。

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    message.text = "Hello Kotlin!"
}
```

多亏Kotlin和Java之间的互操作性，我们可以在Kotlin中像操作属性一样去操作Java库中的getter/setter方法。我们之后再去讲解属性，但是我想提醒的是，我们可以使用 `message.text` 来代替 `message.setText` 。编译器将会把它转换成一般的Java代码，所以这样使用是没有任何性能开销的。

现在运行这个app，并且它是正常运行的。检查TextView是否是显示的新的内容。如果你有疑问或者想查看代码，请在Kotlin for Android Developers repository查看。每个章节只要修改了代码，我都会进行提交，所以一定要检查所有的代码变

化。

下一章会覆盖你在转换之后的MainActivity所看到的新的东西。一旦你理解了Java和Kotlin之间的细微的变化，你将能更容易独立写新的代码了。

类和函数

Kotlin中的类遵循一个简单的结构。尽管与Java有一点细微的差别。你可以使用try.kotlinlang.org在不需要一个真正的项目和不需要部署到机器的前提下来测试一些简单的代码范例。

怎么定义一个类

如果你想定义一个类，你只需要使用 `class` 关键字。

```
class MainActivity{

}
```

它有一个默认唯一的构造器。我们会在以后的课程中学习在特殊的情况下创建其它额外的构造器，但是请记住大部分情况下你只需要这个默认的构造器。你只需要在类名后面写上它的参数。如果这个类没有任何内容可以省略大括号：

```
class Person(name: String, surname: String)
```

那么构造函数的函数体在哪呢？你可以写在 `init` 块中：

```
class Person(name: String, surname: String) {
    init{
        ...
    }
}
```

类继承

默认任何类都是基础继承自 `Any` （与java中的 `Object` 类似），但是我们可以继承其它类。所有的类默认都是不可继承的（final），所以我们只能继承那些明确声明 `open` 或者 `abstract` 的类：

```
open class Animal(name: String)
class Person(name: String, surname: String) : Animal(name)
```

当我们只有单个构造器时，我们需要在从父类继承下来的构造器中指定需要的参数。这是用来替换Java中的 `super` 调用的。

函数

函数（我们Java中的方法）可以使用 `fun` 关键字就可以定义:

```
fun onCreate(savedInstanceState: Bundle?) {
}
```

如果你没有指定它的返回值，它就会返回 `Unit` ，与Java中的 `void` 类似，但是 `Unit` 是一个真正的对象。你当然也可以指定任何其它的返回类型：

```
fun add(x: Int, y: Int) : Int {
    return x + y
}
```

> 小提示：分号不是必须的
>
> > 就像你在上面的例子中看到的那样，我在每句的最后没有使用分号。当然你也可以使用分号，分号不是必须的，而且不使用分号是一个不错的实践。当你这么做了，你会发现这节约了你很多时间。

然而如果返回的结果可以使用一个表达式计算出来，你可以不使用括号而是使用等号：

```
fun add(x: Int,y: Int) : Int = x + y
```

构造方法和函数参数

Kotlin中的参数与Java中有些不同。如你所见，我们先写参数的名字再写它的类型：

```
fun add(x: Int, y: Int) : Int {
    return x + y
}
```

我们可以给参数指定一个默认值使得它们变得可选，这是非常有帮助的。这里有一个例子，在Activity中创建了一个函数用来toast一段信息：

```
fun toast(message: String, length: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(this, message, length).show()
}
```

如你所见，第二个参数（length）指定了一个默认值。这意味着你调用的时候可以传入第二个值或者不传，这样可以避免你需要的重载函数：

```
toast("Hello")
toast("Hello", Toast.LENGTH_LONG)
```

这个与下面的Java代码是一样的：

```
void toast(String message){
}

void toast(String message, int length){
    Toast.makeText(this, message, length).show();
}
```

这跟你想象的一样复杂。再看看这个例子：

```
fun niceToast(message: String,
                tag: String = javaClass<MainActivity>().getSimpl
eName(),
                length: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(this, "[$className] $message", length).show()
}
```

我增加了第三个默认值是类名的tag参数。如果在Java中总数开销会以几何增长。现在可以通过以下方式调用：

```
toast("Hello")
toast("Hello", "MyTag")
toast("Hello", "MyTag", Toast.LENGTH_SHORT)
```

而且甚至还有其它选择，因为你可以使用参数名字来调用，这表示你可以通过在值前写明参数名来传入你希望的参数：

```
toast(message = "Hello", length = Toast.LENGTH_SHORT)
```

> 小提示：String模版
>
> > 你可以在String中直接使用模版表达式。它可以帮助你很简单地在静态值和变量的基础上编写复杂的String。在上面的例子中，我使用了"[$className] $message"。
> >
> > 如你所见，任何时候你使用一个 `$` 符号就可以插入一个表达式。如果这个表达式有一点复杂，你就需要使用一对大括号括起来："Your name is ${user.name}"。

编写你的第一个类

我们已经有了`MainActivity.kt`类。这个Activity会展示下周一系列的天气预报，所有它的layout需要有些改变。

创建一个layout

显示天气预报的列表我们使用 `RecyclerView` ，所以你需要在 `build.gradle` 中增加一个新的依赖：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile "com.android.support:appcompat-v7:$support_version"
    compile "com.android.support:recyclerview-v7:$support_versio
n" ...
}
```

然后， `activity_main.xml` 如下：

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/a
ndroid"
             android:layout_width="match_parent"
             android:layout_height="match_parent">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/forecast_list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</FrameLayout>
```

在 `Mainactivity.kt` 中删除掉之前用来测试的能正常运行的所有代码（现在应该会提示错误）。暂且我们使用老的 `findViewByid()` 的方式：

```
val forecastList = findViewById(R.id.forecast_list) as RecyclerV
iew
forecastList.layoutManager = LinearLayoutManager(this)
```

如你所见，我们定义类一个变量并转型为 `RecyclerView` 。这里与Java有点不同，我们会在下一章分析这些不同之处。 `LayoutManager` 会通过属性的方式被设置，而不是通过setter，这个layout已经足够显示一个列表了。

> 对象实例化
>
> > 对象实例化也是与Java中有些不同。如你所见，我们去掉了 `new` 关键字。这时构造函数仍然会被调用，但是我们省略了宝贵的四个字符。 `LinearLayoutManager(this)` 创建了一个对象的实例。

The Recycler Adapter

我们同样需要一个 `RecyclerView` 的Adapter。之前我在我博客中讨论过 `RecyclerView` ，如果你以前没有使用过，它可能会有所帮助。

`RecyclerView` 中所使用到的布局现在只需要一个 `TextView` ，我会手动去创建这个简单的文本列表。增加一个名为 `ForecastListAdapter.kt` 的Kotlin文件，包括如下代码：

```
class ForecastListAdapter(val items: List<String>) :
        RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
 Int): ViewHolder {
        return ViewHolder(TextView(parent.context))
    }
    override fun onBindViewHolder(holder: ViewHolder, position:
Int) {
        holder.textView.text = items.[position]
    }

    override fun getItemCount(): Int = items.size

    class ViewHolder(val textView: TextView) : RecyclerView.View
Holder(textView)
}
```

又是如此，我们可以像访问属性一样访问context和text。你可以保持以往那样操作（使用getters和setters）,但是你会得到一个编译器的警告。如果你还是倾向于Java中的使用方式，这个检查可以被关闭。但是一旦你使用上了这种属性调用的方式你就会爱上它，而且它也节省了额外的字符总量。

回到 `MainActivity` ，现在简单地创建一系列的String放入List中，然后使用创建分配Adapter实例。

```
private val items = listOf(
    "Mon 6/23 - Sunny - 31/17",
    "Tue 6/24 - Foggy - 21/8",
    "Wed 6/25 - Cloudy - 22/17",
    "Thurs 6/26 - Rainy - 18/11",
    "Fri 6/27 - Foggy - 21/10",
    "Sat 6/28 - TRAPPED IN WEATHERSTATION - 23/18",
    "Sun 6/29 - Sunny - 20/7"
    )

override fun onCreate(savedInstanceState: Bundle?) {
    ...
    val forecastList = findViewById(R.id.forecast_list) as Recyc
lerView
    forecastList.layoutManager = LinearLayoutManager(this)
    forecastList.adapter = ForecastListAdapter(items)
}
```

> List的创建
>
> > 尽管我会在本书后面来对Collection进行讲解，但是我现在仅仅简单地解释你可以通过使用一个函数 `listOf` 创建一个常量的List（很快我们就会讲到的 `immutable` ）。它接收一个任何类型的 `vararg` （可变长的参数），它会自动推断出结果的类型。
> >
> > 还有很多其它的函数可以选择，比如 `setOf` ， `arrayListOf` 或者 `hashSetOf` 。

为了优化项目的结构，我也移动了一些类到新的包里面。

我们在上面很简短的代码中看到了很多新的东西，所以我会在下一章讲到它们。我们现在必须要学习一些比如基本类型，变量，属性等比较重要的概念才能继续下去。

变量和属性

在Kotlin中，一切都是对象。没有像Java中那样的原始基本类型。这个是非常有帮助的，因为我们可以使用一致的方式来处理所有的可用的类型。

基本类型

当然，像integer，float或者boolean等类型仍然存在，但是它们全部都会作为对象存在的。基本类型的名字和它们工作方式都是与Java非常相似的，但是有一些不同之处你可能需要考虑到：

- 数字类型中不会自动转型。举个例子，你不能给 `Double` 变量分配一个 `Int` 。必须要做一个明确的类型转换，可以使用众多的函数之一：

```
val i:Int=7
val d: Double = i.toDouble()
```

- 字符（Char）不能直接作为一个数字来处理。在需要时我们需要把他们转换为一个数字：

```
val c:Char='c'
val i: Int = c.toInt()
```

- 位运算也有一点不同。在Android中，我们经常在 `flags` 中使用“或”，所以我使用"and"和"or来举例：

```
// Java
int bitwiseOr = FLAG1 | FLAG2;
int bitwiseAnd = FLAG1 & FLAG2;
```

```
// Kotlin
val bitwiseOr = FLAG1 or FLAG2
val bitwiseAnd = FLAG1 and FLAG2
```

> 还有很多其他的位操作符，比如 `sh1` , `shs` , `ushr` , `xor` 或 `inv` 。当我们需要的时候，可以在Kotlin官网查看。

- 字面上可以写明具体的类型。这个不是必须的，但是一个通用的Kotlin实践时省略变量的类型（我们马上就能看到），所以我们可以让编译器自己去推断出

具体的类型。

```
val i = 12 // An Int
val iHex = 0x0f // 一个十六进制的Int类型
val l = 3L // A Long
val d = 3.5 // A Double
val f = 3.5F // A Float
```

- 一个String可以像数组那样访问，并且被迭代：

```
val s = "Example"
val c = s[2] // 这是一个字符'a'
// 迭代String
val s = "Example"
for(c in s){
    print(c)
}
```

变量

变量可以很简单地定义成可变(`var`)和不可变（ `val` ）的变量。这个与Java中使用的 `final` 很相似。但是不可变在Kotlin（和其它很多现代语言）中是一个很重要的概念。

一个不可变对象意味着它在实例化之后就不能再去改变它的状态了。如果你需要一个这个对象修改之后的版本，那就会再创建一个新的对象。这个让编程更加具有健壮性和预估性。在Java中，大部分的对象是可变的，那就意味着任何可以访问它这个对象的代码都可以去修改它，从而影响整个程序的其它地方。

不可变对象也可以说是线程安全的，因为它们无法去改变，也不需要去定义访问控制，因为所有线程访问到的对象都是同一个。

所以在Kotlin中，如果我们想使用不可变性，我们编码时思考的方式需要有一些改变。一个重要的概念是：尽可能地使用 **`val`** 。除了个别情况（特别是在Android中，有很多类我们是不会去直接调用构造函数的），大多数时候是可以的。

之前提到的另一件事是我们通常不需要去指明类的类型，它会自动从后面分配的语句中推断出来，这样可以让代码更加清晰和快速修改。我们在前面已经有了一些例子：

```
val s = "Example" // A String
val i = 23 // An Int
val actionBar = supportActionBar // An ActionBar in an Activity
context
```

如果我们需要使用更多的范型类型，则需要指定：

```
val a: Any = 23
val c: Context = activity
```

属性

属性与Java中的字段是相同的，但是更加强大。属性做的事情是字段加上getter加上setter。我们通过一个例子来比较他们的不同之处。这是Java中字段安全访问和修改所需要的代码：

```
public class Person {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
...
Person person = new Person();
person.setName("name");
String name = person.getName();
```

在Kotlin中，只需要一个属性就可以了：

```
public class Person {
    var name: String = ""
}

...

val person = Person()
person.name = "name"
val name = person.name
```

如果没有任何指定，属性会默认使用getter和setter。当然它也可以修改为你自定义的代码，并且不修改存在的代码：

```
public classs Person {
    var name: String = ""
        get() = field.toUpperCase()
        set(value){
            field = "Name: $value"
        }
}
```

如果需要在getter和setter中访问这个属性自身的值，它需要创建一个 `backing field` 。可以使用 `field` 这个预留字段来访问，它会被编译器找到正在使用的并自动创建。需要注意的是，如果我们直接调用了属性，那我们会使用setter和getter而不是直接访问这个属性。 `backing field` 只能在属性访问器内访问。

就如在前面章节提到的，当操作Java代码的时候，Kotlin将允许使用属性的语法去访问在Java文件中定义的getter/setter方法。编译器会直接链接到它原始的getter/setter方法。所以当我们直接访问属性的时候不会有性能开销。

Anko是什么？

Anko是JetBrains开发的一个强大的库。它主要的目的是用来替代以前XML的方式来使用代码生成UI布局。这是一个很有趣的特性，我推荐你可以尝试下，但是我在这个项目中暂时不使用它。对于我（可能是由于多年的UI绘制经验）来说使用XML更容易一些，但是你会喜欢那种方式的。

然而，这个不是我们能在这个库中得到的唯一一个功能。Anko包含了很多的非常有帮助的函数和属性来避免让你写很多的模版代码。我们将会通过本书见到很多例子，但是你应该快速地认识到这个库帮你解决了什么样的问题。

尽管Anko是非常有帮助的，但是我建议你要理解这个背后到底做了什么。你可以在任何时候使用`ctrl + 点击`（Windows）或者`cmd + 点击`（Mac）的方式跳转到Anko的源代码。Anko的实现方式对学习大部分的Kotlin语言都是非常有帮助的。

开始使用Anko

在之前，让我们来使用Anko来简化一些代码。就像你将看到的，任何时候你使用了Anko库中的某些东西，它们都会以属性名、方法等方式被导入。这是因为Anko使用了扩展函数在Android框架中增加了一些新的功能。我们将会在以后看到扩展函数是什么，怎么去编写它。

在 `MainActivity:onCreate` ，一个Anko扩展函数可以被用来简化获取一个RecyclerView：

```
val forecastList: RecyclerView = find(R.id.forecast_list)
```

我们现在还不能使用库中更多的东西，但是Anko能帮助我们简化代码，比如，实例化Intent，Activity之间的跳转，Fragment的创建，数据库的访问，Alert的创建……我们将会在实现这个App的过程中学习到很多有趣的例子。

扩展函数

扩展函数数是指在一个类上增加一种新的行为，甚至我们没有这个类代码的访问权限。这是一个在缺少有用函数的类上扩展的方法。在Java中，通常会实现很多带有static方法的工具类。Kotlin中扩展函数的一个优势是我们不需要在调用方法的时候把整个对象当作参数传入。扩展函数表现得就像是属于这个类的一样，而且我们可以使用 `this` 关键字和调用所有public方法。

举个例子，我们可以创建一个toast函数，这个函数不需要传入任何context，它可以被任何Context或者它的子类调用，比如Activity或者Service：

```
fun Context.toast(message: CharSequence, duration: Int = Toast.L
ENGTH_SHORT) {
    Toast.makeText(this, message, duration).show()
}
```

这个方法可以在Activity内部直接调用：

```
toast("Hello world!")
toast("Hello world!", Toast.LENGTH_LONG)
```

当然，Anko已经包括了自己的toast扩展函数，跟上面这个很相似。Anko提供了一些针对 `CharSequence` 和 `resource` 的函数，还有两个不同的toast和longToast方法：

```
toast("Hello world!")
longToast(R.id.hello_world)
```

扩展函数也可以是一个属性。所以我们可以通过相似的方法来扩展属性。下面的例子展示了使用他自己的getter/setter生成一个属性的方式。Kotlin由于互操作性的特性已经提供了这个属性，但理解扩展属性背后的思想是一个很不错的练习：

```
public var TextView.text: CharSequence
    get() = getText()
    set(v) = setText(v)
```

扩展函数并不是真正地修改了原来的类，它是以静态导入的方式来实现的。扩展函数可以被声明在任何文件中，因此有个通用的实践是把一系列有关的函数放在一个新建的文件里。

这是Anko功能背后的魔法。现在通过以上，你也可以自己创建你的魔法。

执行一个请求

对于感受我们要实现的想法而言，我们目前的文本是很好开始，但是现在是时候去请求一些显示在RecyclerView上的真正的数据了。我们将会使用OpenWeatherMap API来获取数据，还有一些普通类来现实这个请求。多亏Kotlin非常强大的互操作性，你可以使用任何你想使用的库，比如用Retrofit来执行服务器请求。当只是执行一个简单的API请求，我们可以不使用任何第三方库来简单地实现。

而且，如你所见，Kotlin提供了一些扩展函数来让请求变得更简单。首先，我们要创建一个新的Request类：

```
public class Request(val url: String) {
    public fun run() {
        val forecastJsonStr = URL(url).readText()
        Log.d(javaClass.simpleName, forecastJsonStr)
    }
}
```

我们的请求很简单地接收一个url，然后读取结果并在logcat上打印json。实现非常简单，因为我们使用 `readText` ，这是Kotlin标准库中的扩展函数。这个方法不推荐结果很大的响应，但是在我们这个例子中已经足够好了。

如果你用这些代码去比较Java，你会发现我们仅使用标准库就节省了大量的代码。比如 `HttpURLConnection` 、 `BufferedReader` 和需要达到相同效果所必要的迭代结果，管理连接状态、reader等部分的代码。很明显，这些就是场景背后函数所作的事情，但是我们却不用关心。

为了可以执行请求，App必须要有Internet权限。所以需要在 `AndroidManifest.xml` 中添加：：

```
<uses-permission android:name="android.permission.INTERNET" />
```

在主线程以外执行请求

如你所知，HTTP请求不被允许在主线程中执行，否则它会抛出异常。这是因为阻塞住UI线程是一个非常差的体验。Android中通用的做法是使用 `AsyncTask` ，但是这些类是非常丑陋的，并且使用它们无任何副作用地实现功能也是非常困难的。如果你使用不小心， `AsyncTasks` 会非常危险，因为当运行到 `postExecute` 时，如果Activity已经被销毁了，这里就会崩溃。

Anko提供了非常简单的DSL来处理异步任务，它满足大部分的需求。它提供了一个基本的 `async` 函数用于在其它线程执行代码，也可以选择通过调用 `uiThread` 的方式回到主线程。在子线程中执行请求如下这么简单：

```
async() {
    Request(url).run()
    uiThread { longToast("Request performed") }
}
```

`UIThread` 有一个很不错的一点就是可以依赖于调用者。如果它是被一个 `Activity` 调用的，那么如果 `activity.isFinishing()` 返回 `true` ，则 `uiThread` 不会执行，这样就不会在Activity销毁的时候遇到崩溃的情况了。

假如你想使用 `Future` 来工作， `async` 返回一个Java `Future` 。而且如果你需要一个返回结果的 `Future` ，你可以使用 `asyncResult` 。

真的很简单，对吧？而且比 `AsyncTasks` 更加具有可读性。现在，我仅仅给请求发送了一个url，来测试我们是否可以正确接收内容，这样我们才能在Activity中把它画出来。我很快会讲到怎么去进行json解析和转换成app中的数据类，但是在我们继续之前，学习什么是数据类也是很重要的。

检查代码并审查url请求和包结构的代码。你可以运行app并且确保你可以在打印的json日志和请求完毕之后的toast。

数据类

数据类是一种非常强大的类，它可以让你避免创建Java中的用于保存状态但又操作非常简单的POJO的模版代码。它们通常只提供了用于访问它们属性的简单的getter和setter。定义一个新的数据类非常简单：

```
data class Forecast(val date: Date, val temperature: Float, val
details: String)
```

额外的函数

通过数据类，我们可以方便地得到很多有趣的函数，一部分是来自属性，我们之前已经讲过（从编写getter和setter函数）：

- equals(): 它可以比较两个对象的属性来确保他们是相同的。
- hashCode(): 我们可以得到一个hash值，也是从属性中计算出来的。
- copy(): 你可以拷贝一个对象，可以根据你的需要去修改里面的属性。我们会在稍后的例子中看到。
- 一系列可以映射对象到变量中的函数。我也很快就会讲到这个。

复制一个数据类

如果我们使用不可修改的对象，就像我们之前讲过的，假如我们需要修改这个对象状态，必须要创建一个新的一个或者多个属性被修改的实例。这个任务是非常重复且不简洁的。

举个例子，如果我们需要修改 `Forecast` 中的 `temperature` （温度），我们可以这么做：

```
val f1 = Forecast(Date(), 27.5f, "Shiny day")
val f2 = f1.copy(temperature = 30f)
```

如上，我们拷贝了第一个forecast对象然后只修改了 `temperature` 的属性而没有修改这个对象的其它状态。

> 当你使用Java类时小心“不可修改性”
>
> > 如果你决定使用不可修改来工作，你需要意识到Java不是根据这种思想来设计的，在某些情况下，我们仍然可以修改这些状态。在上一个例子中，你还是可以访问 `Date` 对象，然后改变它的值。有个简单（不安全）的方法是记住所有需要修改状态的对象作为一个规则，然后必要的时候去拷贝一份。
> >
> > 另外一个方法是封装这些类。你可以创建一个 `ImmutableDate` 类，它封装了 `Date` 并且不允许去修改它们的状态。决定哪种方式取决于你。本书中，我不会对不可修改性太限制，所以我不会去为一些危险的类去创建一个封装类。

映射对象到变量中

映射对象的每一个属性到一个变量中，这个过程就是我们知道的多声明。这就是为什么会有 `componentX` 函数被自动创建。使用上面的 `Forecast` 类举个例子：

```
val f1 = Forecast(Date(), 27.5f, "Shiny day")
val (date, temperature, details) = f1
```

上面这个多声明会被编译成下面的代码：

```
val date = f1.component1()
val temperature = f1.component2()
val details = f1.component3()
```

这个特性背后的逻辑是非常强大的，它可以在很多情况下帮助我们简化代码。举个例子， `Map` 类含有一些扩展函数的实现，允许它在迭代时使用key和value：

```
for ((key, value) in map) {
    Log.d("map", "key:$key, value:$value")
}
```

转换json到数据类

我们现在知道怎么去创建一个数据类，那我们开始准备去解析数据。在 `date` 包中，创建一个名为 `ResponseClasses.kt` 新的文件，如果你打开第8章中的url，你可以看到json文件整个结构。它的基本组成包括一个城市，一个系列的天气预报，这个城市有id，名字，所在的坐标。每一个天气预报有很多信息，比如日期，不同的温度，和一个由描述和图标的id。

在我们当前的UI中，我们不会去使用所有的这些数据。我们会解析所有到类里面，因为可能会在以后某些情况下会用到。以下就是我们需要使用到的类：

```
data class ForecastResult(val city: City, val list: List<Forecast>)
data class City(val id: Long, val name: String, val coord: Coordinates,
                val country: String, val population: Int)
data class Coordinates(val lon: Float, val lat: Float)
data class Forecast(val dt: Long, val temp: Temperature, val pressure: Float,
                    val humidity: Int, val weather: List<Weather>,
                    val speed: Float, val deg: Int, val clouds: Int,
                    val rain: Float)
data class Temperature(val day: Float, val min: Float, val max: Float,
                       val night: Float, val eve: Float, val morn: Float)
data class Weather(val id: Long, val main: String, val description: String,
                   val icon: String)
```

当我们使用Gson来解析json到我们的类中，这些属性的名字必须要与json中的名字一样，或者可以指定一个 `serialised name` （序列化名称）。一个好的实践是，大部分的软件结构中会根据我们app中布局来解耦成不同的模型。所以我喜欢使用

声明简化这些类，因为我会在app其它部分使用它之前解析这些类。属性名称与json结果中的名字是完全一样的。

现在，为了返回被解析后的结果，我们的 `Request` 类需要进行一些修改。它将仍然只接收一个城市的 `zipcode` 作为参数而不是一个完整的url，因此这样变得更加具有可读性。现在，我会把这个静态的url放在一个 `companion object` （伴随对象）中。如果我们之后还要对该API增加更多请求，我们需要提取它。

> **Companion objects**
>
> > Kotlin允许我们去定义一些行为与静态对象一样的对象。尽管这些对象可以用众所周知的模式来实现，比如容易实现的单例模式。
> >
> > 我们需要一个类里面有一些静态的属性、常量或者函数，我们可以使用 `companion objecvt` 。这个对象被这个类的所有对象所共享，就像Java中的静态属性或者方法。

以下是最后的代码：

```
public class ForecastRequest(val zipCode: String) {
    companion object {
        private val APP_ID = "15646a06818f61f7b8d7823ca833e1ce"
        private val URL = "http://api.openweathermap.org/data/2.
5/" +
                "forecast/daily?mode=json&units=metric&cnt=7"
        private val COMPLETE_URL = "$URL&APPID=$APP_ID&q="
    }

    fun execute(): ForecastResult {
        val forecastJsonStr = URL(COMPLETE_URL + zipCode).readTe
xt()
        return Gson().fromJson(forecastJsonStr, ForecastResult::
class.java)
    }
}
```

记得在 `build.gradle` 中增加你需要的Gson依赖：

```
compile "com.google.code.gson:gson:2.4"
```

构建domain层

我们现在创建一个新的包作为 `domain` 层。这一层中会包含一些 `Commands` 的实现来为app执行任务。

首先，必须要定义一个 `Command` ：

```
public interface Command<T> {
    fun execute(): T
}
```

这个command会执行一个操作并且返回某种类型的对象，这个类型可以通过范型被指定。你需要知道一个有趣的概念，一切**kotlin**函数都会返回一个值。如果没有指定，它就默认返回一个 `Unit` 类。所以如果我们想让Command不返回数据，我们可以指定它的类型为Unit。

Kotlin中的接口比Java（Java 8以前）中的强大多了，因为它们可以包含代码。但是我们现在不需要更多的代码，以后的章节中会仔细讲这个话题。

第一个command需要去请求天气预报结构然后转换结果为domain类。下面这些类会在domain类中被定义：

```
data class ForecastList(val city: String, val country: String,
                        val dailyForecast:List<Forecast>)

data class Forecast(val date: String, val description: String, v
al high: Int,
                    val low: Int)
```

当更多的功能被增加，这些类可能会需要在以后被审查。但是现在这些类对我们来说已经足够了。

这些类必须从数据映射到我们的domain类，所以我接下来需要创建一个 `DataMapper` ：

```
public class ForecastDataMapper {
    fun convertFromDataModel(forecast: ForecastResult): Forecast
List {
        return ForecastList(forecast.city.name, forecast.city.co
untry,
                convertForecastListToDomain(forecast.list))
    private fun convertForecastListToDomain(list: List<Forecast>)
:
            List<ModelForecast> {
        return list.map { convertForecastItemToDomain(it) }
    }
    private fun convertForecastItemToDomain(forecast: Forecast):
 ModelForecast {
        return ModelForecast(convertDate(forecast.dt),
                forecast.weather[0].description, forecast.temp.m
ax.toInt(),
                forecast.temp.min.toInt())
}
    private fun convertDate(date: Long): String {
        val df = DateFormat.getDateInstance(DateFormat.MEDIUM, L
ocale.getDefault())
        return df.format(date * 1000)
    }
}
```

当我们使用了两个相同名字的类，我们可以给其中一个指定一个别名，这样我们就不需要写完整的包名了：

```
import com.antonioleiva.weatherapp.domain.model.Forecast as Mode
lForecast
```

这些代码中另一个有趣的是我们从一个forecast list中转换为domain model的方法：

```
return list.map { convertForecastItemToDomain(it) }
```

这一条语句，我们就可以循环这个集合并且返回一个转换后的新的List。Kotlin在List中提供了很多不错的函数操作符，它们可以在这个List的每个item中应用这个操作并且任何方式转换它们。对比Java 7，这是Kotlin其中一个强大的功能。我们很快就会查看所有不同的操作符。知道它们的存在是很重要的，因为它们要方便得多，并可以节省很多时间和模版。

现在，编写命令前的准备就绪：

```
class RequestForecastCommand(val zipCode: String) :
                Command<ForecastList> {
    override fun execute(): ForecastList {
        val forecastRequest = ForecastRequest(zipCode)
        return ForecastDataMapper().convertFromDataModel(
                forecastRequest.execute())
    }
}
```

在UI中绘制数据

`MainActivity` 中的代码有些小的改动，因为现在有真实的数据需要填充到adapter中。异步调用需要被重写成：

```
async() {
    val result = RequestForecastCommand("94043").execute()
    uiThread{
        forecastList.adapter = ForecastListAdapter(result)
    }
}
```

Adapter也需要被修改：

```
class ForecastListAdapter(val weekForecast: ForecastList) :
        RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
 Int):
            ViewHolder? {
        return ViewHolder(TextView(parent.getContext()))
    }

    override fun onBindViewHolder(holder: ViewHolder,
            position: Int) {
        with(weekForecast.dailyForecast[position]) {
            holder.textView.text = "$date - $description - $high
/$low"
    }
}
    override fun getItemCount(): Int = weekForecast.dailyForecas
t.size

    class ViewHolder(val textView: TextView) : RecyclerView.View
Holder(textView)
}
```

> **with函数**
>
> > with是一个非常有用的函数，它包含在Kotlin的标准库中。它接收一个对象和一个扩展函数作为它的参数，然后使这个对象扩展这个函数。这表示所有我们在括号中编写的代码都是作为对象（第一个参数）的一个扩展函数，我们可以就像作为this一样使用所有它的public方法和属性。当我们针对同一个对象做很多操作的时候这个非常有利于简化代码。

在这一章中有很多新的代码加入，所以检出库中的代码吧。

操作符重载

Kotin有一些固定数量象征性的操作符，我们可以在任何类中很容易地使用它们。方法是创建一个方法，方法名为保留的操作符关键字，这样就可以让这个操作符的行为映射到这个方法。重载这些操作符可以增加代码可读性和简洁性。

操作符表

这里你可以看见一系列包括 `操作符` 和 `对应方法` 的表。对应方法必须在指定的类中通过各种可能性被实现。

一元操作符

操作符	函数
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()
a++	a.inc()
a--	a.dec()

二元操作符

操作符	函数
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.mod(b)
a..b	a.rangeTo(b)
a in b	b.contains(a)
a !in b	!b.contains(a)
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.modAssign(b)

数组操作符

操作符	函数
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)

等于操作符

操作符	函数
a == b	a?.equals(b) ?: b === null
a != b	!(a?.equals(b) ?: b === null)

相等操作符有一点不同，为了达到正确合适的相等检查做了更复杂的转换，因为要得到一个确切的函数结构比较，不仅仅是指定的名称。方法必须要如下准确地被实现：

```
operator fun equals(other: Any?): Boolean
```

操作符 `===` 和 `!==` 用来做身份检查（它们分别是Java中的 `==` 和 `!=` ），并且它们不能被重载。

函数调用

方法	调用
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

例子

你可以想象，Kotlin List是实现了数组操作符的，所以我们可以像Java中的数组一样访问List的每一项。除此之外：在可修改的List中，每一项也可以用一个简单的方式被直接设置：

```
val x = myList[2]
myList[2] = 4
```

如果你还记得，我们有一个叫ForecastList的数据类，它是由很多其他额外的信息组成的。有趣的是可以直接访问它的每一项而不是请求内部的list得到某一项。做一个完全不相关的事情，我要去实现一个 `size()` 方法，它能稍微能简化一点当前的Adapter：

```
data class ForecastList(val city: String, val country: String,
                        val dailyForecast: List<Forecast>) {
    operator fun get(position: Int): Forecast = dailyForecast[po
sition]
    fun size(): Int = dailyForecast.size
}
```

它会使我们的 `onBindViewHolder` 更简单一点：

```
override fun onBindViewHolder(holder: ViewHolder,
        position: Int) {
    with(weekForecast[position]) {
        holder.textView.text = "$date - $description - $high/$lo
w"
    }
}
```

当然还有 `getItemCount()` 方法：

```
override fun getItemCount(): Int = weekForecast.size()
```

扩展函数中的操作符

我们不需要去扩展我们自己的类，但是我需要去使用扩展函数扩展我们已经存在的类来让第三方的库能提供更多的操作。几个例子，我们可以去像访问List的方式去访问 `ViewGroup` 的view：

```
operator fun ViewGroup.get(position: Int): View = getChildAt(position)
```

现在真的可以非常简单地从一个 `ViewGroup` 中通过position得到一个view：

```
val container: ViewGroup = find(R.id.container)
val view = container[2]
```

别忘了去Kotlin for Android developers repository去查看这些代码。

使Forecast list可点击

作为一个真正的app，当前列表的每一个item布局应该做一些工作。第一件事就是创建一个合适的XML，能符合我们的需要就行。我们希望显示一个图标，日期，描述以及最高和最低温度。所以让我们创建一个名为 `item_forecast.xml` 的layout：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/spacing_xlarge"
    android:background="?attr/selectableItemBackground"
    android:gravity="center_vertical"
    android:orientation="horizontal">

    <ImageView
        android:id="@+id/icon"
        android:layout_width="48dp"
        android:layout_height="48dp"
        tools:src="@mipmap/ic_launcher"/>

    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_marginLeft="@dimen/spacing_xlarge"
        android:layout_marginRight="@dimen/spacing_xlarge"
        android:orientation="vertical">

    <TextView
        android:id="@+id/date"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAppearance="@style/TextAppearance.AppCompat.
```

```
Medium"
        tools:text="May 14, 2015"/>

    <TextView
        android:id="@+id/description"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAppearance="@style/TextAppearance.AppCompat.
Caption"
        tools:text="Light Rain"/>

    </LinearLayout>
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal"
        android:orientation="vertical">

    <TextView
        android:id="@+id/maxTemperature"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="@style/TextAppearance.AppCompat.
Medium"
        tools:text="30"/>

    <TextView
        android:id="@+id/minTemperature"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="@style/TextAppearance.AppCompat.
Caption"
        tools:text="15"/>

    </LinearLayout>
</LinearLayout>
```

Domain model和数据映射时必须生成完整的图标uil，所以我们可以这样去加载它：

```
data class Forecast(val date: String, val description: String,
                    val high: Int, val low: Int, val iconUrl: St
ring)
```

在 `ForecastDataMapper` 中：

```
private fun convertForecastItemToDomain(forecast: Forecast): Mod
elForecast {
    return ModelForecast(convertDate(forecast.dt),
            forecast.weather[0].description, forecast.temp.max.t
oInt(),
            forecast.temp.min.toInt(), generateIconUrl(forecast.
weather[0].icon))
}

private fun generateIconUrl(iconCode: String): String
        = "http://openweathermap.org/img/w/$iconCode.png"
```

我们从第一个请求中得到图标的code，用来组成完成的图标url。加载图片最简单的方式是使用图片加载库。`Picasso` 是一个不错的选择。它需要加到 `build.gradle` 的依赖中：

```
compile "com.squareup.picasso:picasso:<version>"
```

如此，Adapter也需要一个大的改动了。还需要一个click listener，我们来定义它：

```
public interface OnItemClickListener {
     operator fun invoke(forecast: Forecast)
}
```

如果你还记得上一课程，当被调用时 `invoke` 方法可以被省略。所以我们来使用它来简化。listener可以被以下两种方式调用：

```
itemClick.invoke(forecast)
itemClick(forecast)
```

`ViewHolder` 将负责去绑定数据到新的View：

```
class ViewHolder(view: View, val itemClick: OnItemClickListener)
 :
                RecyclerView.ViewHolder(view) {
    private val iconView: ImageView
    private val dateView: TextView
    private val descriptionView: TextView
    private val maxTemperatureView: TextView
    private val minTemperatureView: TextView

    init {
        iconView = view.find(R.id.icon)
        dateView = view.find(R.id.date)
        descriptionView = view.find(R.id.description)
        maxTemperatureView = view.find(R.id.maxTemperature)
        minTemperatureView = view.find(R.id.minTemperature)
    }

    fun bindForecast(forecast: Forecast) {
        with(forecast) {
            Picasso.with(itemView.ctx).load(iconUrl).into(iconVi
ew)
            dateView.text = date
            descriptionView.text = description
            maxTemperatureView.text = "${high.toString()}"
            minTemperatureView.text = "${low.toString()}"
            itemView.setOnClickListener { itemClick(forecast) }
        }
    }
}
```

现在Adapter的构造方法接收一个 `itemClick` 。创建和绑定数据也是更简单：

```
public class ForecastListAdapter(val weekForecast: ForecastList,
         val itemClick: ForecastListAdapter.OnItemClickListener)
 :
        RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
 Int):
            ViewHolder {
        val view = LayoutInflater.from(parent.ctx)
            .inflate(R.layout.item_forecast, parent, false)
        return ViewHolder(view, itemClick)
    }

    override fun onBindViewHolder(holder: ViewHolder, position:
Int) {
        holder.bindForecast(weekForecast[position])
    }
    ...
}
```

如果你使用了上面这些代码，`parent.ctx` 不会被编译成功。Anko提供了大量的扩展函数来让Android编程更简单。举个例子，activitys、fragments以及其它包含了 `ctx` 这个属性，通过 `ctx` 这个属性来返回context，但是在View中缺少这个属性。所以我们要创建一个新的名叫 `ViewExtensions.kt` 文件来代替 `ui.utils`，然后增加这个扩展属性：

```
val View.ctx: Context
    get() = context
```

从现在开始，任何View都可以使用这个属性了。这个不是必须的，因为你可以使用扩展的context属性，但是我觉得如果我们使用 `ctx` 的话在其它类中也会更有连贯性。而且，这是一个很好的怎么去使用扩展属性的例子。

最后，MainActivity调用setAdapter，最后结果是这样的：

```
forecastList.adapter = ForecastListAdapter(result,
        object : ForecastListAdapter.OnItemClickListener{
            override fun invoke(forecast: Forecast) {
                toast(forecast.date)
            }
        })
```

如你所见，创建一个匿名内部类，我们去创建了一个实现了刚刚创建的接口的对象。看起来不是很好，对吧？这是因为我们还没开始试使用另一个强大的函数式编程的特性，但是你将会在下一章中学习到怎么去把这些代码转换得更简单。

去代码库中更新新的代码。UI开始看起来更好了。

Lambdas

Lambda表达式是一种很简单的方法，去定义一个匿名函数。Lambda是非常有用的，因为它们避免我们去写一些包含了某些函数的抽象类或者接口，然后在类中去实现它们。在Kotlin，我们把一个函数作为另一个函数的参数。

简化setOnClickListener()

我们用Android中非常典型的例子去解释它是怎么工作的：`View.setOnClickListener()` 方法。如果我们想用Java的方式去增加点击事件的回调，我首先要编写一个 `OnClickListener` 接口：

```
public interface OnClickListener {
    void onClick(View v);
}
```

然后我们要编写一个匿名内部类去实现这个接口：

```
view.setOnClickListener(new OnClickListener(){
    @Override
    public void onClick(View v) {
        Toast.makeText(v.getContext(), "Click", Toast.LENGTH_SHO
RT).show();
    }
})
```

我们将把上面的代码转换成Kotlin（使用了Anko的toast函数）：

```
view.setOnClickListener(object : OnClickListener {
    override fun onClick(v: View) {
        toast("Click")
    }
}
```

很幸运的是，Kotlin允许Java库的一些优化，Interface中包含单个函数可以被替代为一个函数。如果我们这么去定义了，它会正常执行：

```
fun setOnClickListener(listener: (View) -> Unit)
```

一个lambda表达式通过参数的形式被定义在箭头的左边（被圆括号包围），然后在箭头的右边返回结果值。在这个例子中，我们接收一个View，然后返回一个Unit（没有东西）。所以根据这种思想，我们可以把前面的代码简化成这样：

```
view.setOnClickListener({ view -> toast("Click")})
```

这是非常棒的简化！当我们定义了一个方法，我们必须使用大括号包围，然后在箭头的左边指定参数，在箭头的右边返回函数执行的结果。如果左边的参数没有使用到，我们甚至可以省略左边的参数：

```
view.setOnClickListener({ toast("Click") })
```

如果这个函数的最后一个参数是一个函数，我们可以把这个函数移动到圆括号外面：

```
view.setOnClickListener() { toast("Click") }
```

并且，最后，如果这个函数只有一个参数，我们可以省略这个圆括号：

```
view.setOnClickListener { toast("Click") }
```

比原始的Java代码简短了5倍多，并且更加容易理解它所做的事情。非常让人影响深刻。

ForecastListAdapter的click listener

在前面一章，我这么艰苦地写了click listener的目的就是更好的在这一章中进行开发。然而现在是时候把你学到的东西用到实践中去了。我们从ForecastListAdapter中删除了listener接口，然后使用lambda代替：

```
public class ForecastListAdapter(val weekForecast: ForecastList,
                                 val itemClick: (Forecast) -> Un
it)
```

这个itemClick函数接收一个 `forecast` 参数然后不返回任何东西。 `ViewHolder` 中也可以这么修改：

```
class ViewHolder(view: View, val itemClick: (Forecast) -> Unit)
```

其它的代码保持不变。仅仅改变 `MainActivity` ：

```
val adapter = ForecastListAdapter(result) { forecast -> toast(fo
recast.date) }
```

我们可以简化最后一句。如果这个函数只接收一个参数，那我们可以使用 `it` 引用，而不用去指定左边的参数。所以我们可以这么做：

```
val adapter = ForecastListAdapter(result) { toast(it.date) }
```

扩展语言

多亏这些改变，我们可以去创建自己的 `builder` 和代码块。我们已经在使用一些有趣的函数，比如 `with` 。如下简单的实现：

```
inline fun <T> with(t: T, body: T.() -> Unit) { t.body() }
```

这个函数接收一个 `T` 类型的对象和一个被作为扩展函数的函数。它的实现仅仅是让这个对象去执行这个函数。因为第二个参数是一个函数，所以我们可以把它放在圆括号外面，所以我们可以创建一个代码块，在这这个代码块中我们可以使用 `this` 和直接访问所有的public的方法和属性：

```
with(forecast) {
    Picasso.with(itemView.ctx).load(iconUrl).into(iconView)
    dateView.text = date
    descriptionView.text = description
    maxTemperatureView.text = "$high"
    minTemperatureView.text = "$low"
    itemView.setOnClickListener { itemClick(this) }
}
```

> 内联函数
>
> > 内联函数与普通的函数有点不同。一个内联函数会在编译的时候被替换掉，而不是真正的方法调用。这在一些情况下可以减少内存分配和运行时开销。举个例子，如果我们有一个函数，只接收一个函数作为它的参数。如果是一个普通的函数，内部会创建一个含有那个函数的对象。另一方面，内联函数会把我们调用这个函数的地方替换掉，所以它不需要为此生成一个内部的对象。

另一个例子：我们可以创建代码块只提供 `Lollipop` 或者更高版本来执行：

```
inline fun supportsLollipop(code: () -> Unit) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
        code()
    }
}
```

它只是检查版本，然后如果满足条件则去执行。现在我们可以这么做：

```
supportsLollipop {
    window.setStatusBarColor(Color.BLACK)
}
```

举个例子，Anko也是基于这个思想来实现 `Android Layout` 的 `DSL` 化。你也可以查看 `Kotlin reference` 中 `使用DSL来编写HTML` 的一个例子。

可见性修饰符

Kotlin中这些修饰符是与我们Java中的使用是有些不同的。在这个语言中默认的修饰符是 `public` ，这节约了很多的时间和字符。但是这里有一个详细的解释关于在Kotlin中不同的可见性修饰符是怎么工作的。

修饰符

private

`private` 修饰符是我们使用的最限制的修饰符。它表示它只能被自己所在的文件可见。所以如果我们给一个类声明为 `private` ，我们就不能在定义这个类之外的文件中使用它。

另一方面，如果我们在一个类里面使用了 `private` 修饰符，那访问权限就被限制在这个类里面了。甚至是继承这个类的子类也不能使用它。

所以一等公民，类、对象、接口……（也就是包成员）如果被定义为 `private` ，那么它们只会对被定义所在的文件可见。如果被定义在了类或者接口中，那它们只对这个类或者接口可见。

protected

这个修饰符只能被用在类或者接口中的成员上。一个包成员不能被定义为 `protected` 。定义在一个成员中，就与Java中的方式一样了：它可以被成员自己和继承它的成员可见（比如，类和它的子类）。

internal

如果是一个定义为 `internal` 的包成员的话，对所在的整个 `module` 可见。如果它是一个其它领域的成员，它就需要依赖那个领域的可见性了。比如，如果我们写了一个 `private` 类，那么它的 `internal` 修饰的函数的可见性就会限制与它所在的这个类的可见性。

我们可以访问同一个 `module` 中的 `internal` 修饰的类，但是不能访问其它 `module` 的。

> 什么是`module`
>
> > 根据Jetbrains的定义，一个`module`应该是一个单独的功能性的单位，它应该是可以被单独编译、运行、测试、debug的。根据我们项目不同的模块，可以在Android Studio中创建不同的`module`。在Eclipse中，这些`module`可以认为是在一个`workspace`中的不同的`project`。

public

你应该可以才想到，这是最没有限制的修饰符。这是默认的修饰符，成员在任何地方被修饰为`public`，很明显它只限制于它的领域。一个定义为`public`的成员被包含在一个`private`修饰的类中，这个成员在这个类以外也是不可见的。

构造器

所有构造函数默认都是 `public` 的，它们类是可见的，可以被其它地方使用。我们也可以使用这个语法来把构造函数修改为 `private` ：

```
class C private constructor(a: Int) { ... }
```

润色我们的代码

我们已经准备好使用 `public` 来进行重构了，但是我们还有很多其它细节需要修改。比如，在 `RequestForecastCommand` 中，我们在构造函数中我们创建的属性 `zipCode` 可以定义为 `private` ：

```
class RequestForecastCommand(private val zipCode: String)
```

所作的事情就是我们创建了一个不可修改的属性zipCode，它的值我们只能去得到，不能去修改它。所以这个不大的改动让代码看起来更加清晰。如果我们在编写类的时候，你觉得某些属性因为是什么原因不能对别人可见，那就把它定义为 `private` 。

而且，在Kotlin中，我们不需要去指定一个函数的返回值类型，它可以让编译器推断出来。举个省略返回值类型的例子：

```
data class ForecastList(...) {
    fun get(position: Int) = dailyForecast[position]
    fun size() = dailyForecast.size()
}
```

我们可以省略返回值类型的典型情景是当我们要给一个函数或者一个属性赋值的时候。而不需要去写代码块去实现。

剩下的修改是相当简单的，你可以在代码库中去同步下来。

Kotlin Android Extensions

另一个Kotlin团队研发的可以让开发更简单的插件是 `Kotlin Android Extensions` 。当前仅仅包括了view的绑定。这个插件自动创建了很多的属性来让我们直接访问XML中的view。这种方式不需要你在开始使用之前明确地从布局中去找到这些views。

这些属性的名字就是来自对应view的id，所以我们取id的时候要十分小心，因为它们将会是我们类中非常重要的一部分。这些属性的类型也是来自XML中的，所以我们不需要去进行额外的类型转换。

`Kotlin Android Extensions` 的一个优点是它不需要在我们的代码中依赖其它额外的库。它仅仅由插件组层，需要时用于生成工作所需的代码，只需要依赖于Kotlin的标准库。

那它背后是怎么工作的？该插件会代替任何属性调用函数，比如获取到view并具有缓存功能，以免每次属性被调用都会去重新获取这个view。需要注意的是这个缓存装置只会在 `Activity` 或者 `Fragment` 中才有效。如果它是在一个扩展函数中增加的，这个缓存就会被跳过，因为它可以被用在 `Activity` 中但是插件不能被修改，所以不需要再去增加一个缓存功能。

怎么去使用Kotlin Android Extensions

如果你还记得，现在项目已经准备好去使用Kotlin Android Extensions。当我们创建这个项目，我们就已经在 `build.gradle` 中增加了这个依赖：

```
buldscript{
    repositories {
        jcenter()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-android-extension
s:$kotlin_version"
    }
}
```

唯一一件需要这个插件做的事情是在类中增加一个特定的"手工" `import` 来使用这个功能。我们有两个方法来使用它：

`Activities` 或者 `Fragments` 的 `Android Extensions`

这是最典型的使用方式。它们可以作为 `activity` 或 `fragment` 的属性是可以被访问的。属性的名字就是XML中对应view的id。

我们需要使用的 `import` 语句以 `kotlin.android.synthetic` 开头，然后加上我们要绑定到Activity的布局XML的名字：

```
import kotlinx.android.synthetic.activity_main.*
```

此后，我们就可以在 `setContentView` 被调用后访问这些view。新的Android Studio版本中可以通过使用 `include` 标签在Activity默认布局中增加内嵌的布局。很重要的一点是，针对这些布局，我们也需要增加手工的import：

```
import kotlinx.android.synthetic.activity_main.*
import kotlinx.android.synthetic.content_main.*
```

`Views` 的 `Android Extensions`

前面说的使用还是有局限性的，因为可能有很多代码需要访问XML中的view。比如，一个自定义view或者一个adapter。举个例子，绑定一个xml中的view到另一个view。唯一不同的就是需要 `import` ：

```
import kotlinx.android.synthetic.view_item.view.*
```

如果我们需要一个adapter，比如，我们现在要从inflater的View中访问属性：

```
view.textView.text = "Hello"
```

重构我们的代码

现在是时候使用 `Kotlin Android Extensions` 来修改我们的代码了。修改相当简单。

我们从 `MainActivity` 开始。我们当前只是使用了 `forecastList` 的RecyclerView。但是我们可以简化一点代码。首先，为 `activity_main` XML增加手工import：

```
import kotlinx.android.synthetic.activity_main.*
```

之前说过，我们使用id来访问views。所以我要修改 `RecyclerView` 的id，不使用下划线，让它更加适合Kotlin变量的名字。XML最后如下：

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/forecastList"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</FrameLayout>
```

然后现在，我们可以不需要 `find` 这一行了：

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    forecastList.layoutManager = LinearLayoutManager(this)
    ...
}
```

这已经是最小的简化了，因为这个布局非常简单。但是 `ForecastListAdapter` 也可以从这个插件中受益。这里你可以使用一个装置来绑定这些属性到view中，它可以帮助我们移除所有 `ViewHolder` 的 `find` 代码。

首先，为 `item_forecast` 增加手工导入：

```
import kotlinx.android.synthetic.item_forecast.view.*
```

然后现在我们可以在 `ViewHolder` 中使用包含在 `itemView` 中的属性。实际上你可以在任何view中使用这些属性，但是很显然如果view不包含要获取的子view就会奔溃。

现在我们可以直接访问view的属性了：

```
class ViewHolder(view: View, val itemClick: (Forecast) -> Unit)
:
        RecyclerView.ViewHolder(view) {
    fun bindForecast(forecast: Forecast) {
        with(forecast){
            Picasso.with(itemView.ctx).load(iconUrl).into(itemVi
ew.icon)
            itemView.date.text = date
            itemView.description.text = description
            itemView.maxTemperature.text = "${high.toString()}"

            itemView.minTemperature.text = "${low.toString()}"
            itemView.onClick { itemClick(forecast) }
        }
    }
}
```

Kotlin Android Extensions插件帮助我们减少了很多模版代码，并且简化了我们访问view的方式。从库中检出最新的代码吧。

Application单例化和属性的Delegated

我们很快要去实现一个数据库，如果我们想要保持我们代码的简洁性和层次性（而不是把所有代码添加到Activity中），我们就要需要有一个更简单的访问application context的方式。

Applicaton单例化

按照我们在Java中一样创建一个单例最简单的方式：

```
class App : Application() {
    companion object {
        private var instance: Application? = null
        fun instance() = instance!!
    }
    override fun onCreate() {
        super.onCreate()
        instance = this
    }
}
```

为了这个Application实例被使用，要记得在 `AndroidManifest.xml` 中增加这个 `App` ：

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme"
    android:name=".ui.App">
    ...
</application>
```

Android有一个问题，就是我们不能去控制很多类的构造函数。比如，我们不能初始化一个非null属性，因为它的值需要在构造函数中去定义。所以我们需要一个可null的变量，和一个返回非null值的函数。我们知道我们一直都有一个 `App` 实例，但是在它调用 `onCreate` 之前我们不能去操作任何事情，所以我们为了安全性，我们假设 `instance()` 函数将会总是返回一个非null的 `app` 实例。

但是这个方案看起来有点不自然。我们需要定义个一个属性（已经有了getter和setter），然后通过一个函数来返回那个属性。我们有其他方法去达到相似的效果么？是的，我们可以通过委托这个属性的值给另外一个类。这个就是我们知道

的 `委托属性` 。

委托属性

我们可能需要一个属性具有一些相同的行为，使用 `lazy` 或者 `observable` 可以被很有趣地实现重用。而不是一次又一次地去声明那些相同的代码，Kotlin提供了一个委托属性到一个类的方法。这就是我们知道的 `委托属性` 。

当我们使用属性的 `get` 或者 `set` 的时候，属性委托的 `getValue` 和 `setValue` 就会被调用。

属性委托的结构如下：

```
class Delegate<T> : ReadWriteProperty<Any?, T> {
    fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return ...
    }

    fun setValue(thisRef: Any?, property: KProperty<*>, value: T)
 {
        ...
    }
}
```

这个T是委托属性的类型。 `getValue` 函数接收一个类的引用和一个属性的元数据。 `setValue` 函数又接收了一个被设置的值。如果这个属性是不可修改（val），就会只有一个 `getValue` 函数。

下面展示属性委托是怎么设置的：

```
class Example {
    var p: String by Delegate()
}
```

它使用了 `by` 这个关键字来指定一个委托对象。

标准委托

在Kotlin的标准库中有一系列的标准委托。它们包括了大部分有用的委托，但是我们也可以创建我们自己的委托。

Lazy

它包含一个lambda，当第一次执行 `getValue` 的时候这个lambda会被调用，所以这个属性可以被延迟初始化。之后的调用都只会返回同一个值。这是非常有趣的特性， 当我们在它们第一次真正调用之前不是必须需要它们的时候。我们可以节省内存，在这些属性真正需要前不进行初始化。

```
class App : Application() {
    val database: SQLiteOpenHelper by lazy {
        MyDatabaseHelper(applicationContext)
    }

    override fun onCreate() {
        super.onCreate()
        val db = database.writableDatabase
    }
}
```

在这个例子中，database并没有被真正初始化，直到第一次调用 `onCreate` 时。在那之后，我们才确保applicationContext存在，并且已经准备好可以被使用了。 `lazy` 操作符是线程安全的。

如果你不担心多线程问题或者想提高更多的性能，你也可以使用 `lazy(LazyThreadSafeMode.NONE){ ... }` 。

Observable

这个委托会帮我们监测我们希望观察的属性的变化。当被观察属性的 `set` 方法被调用的时候，它就会自动执行我们指定的lambda表达式。所以一旦该属性被赋了新的值，我们就会接收到被委托的属性、旧值和新值。

```
class ViewModel(val db: MyDatabase) {
    var myProperty by Delegates.observable("") {
        d, old, new ->
        db.saveChanges(this, new)
    }
}
```

这个例子展示了，一些我们需要关心的ViewMode，每次值被修改了，就会保存它们到数据库。

Vetoable

这是一个特殊的`observable`，它让你决定是否这个值需要被保存。它可以被用于在真正保存之前进行一些条件判断。

```
var positiveNumber = Delegates.vetoable(0) {
    d, old, new ->
    new >= 0
}
```

上面这个委托只允许在新的值是正数的时候执行保存。在lambda中，最后一行表示返回值。你不需要使用return关键字（实质上不能被编译）。

Not Null

有时候我们需要在某些地方初始化这个属性，但是我们不能在构造函数中确定，或者我们不能在构造函数中做任何事情。第二种情况在Android中很常见：在Activity、fragment、service、receivers……无论如何，一个非抽象的属性在构造函数执行完之前需要被赋值。为了给这些属性赋值，我们无法让它一直等待到我们希望给它赋值的时候。我们至少有两种选择方案。

第一种就是使用可null类型并且赋值为null，直到我们有了真正想赋的值。但是我们就需要在每个地方不管是否是null都要去检查。如果我们确定这个属性在任何我们使用的时候都不会是null，这可能会使得我们要编写一些必要的代码了。

第二种选择是使用 `notNull` 委托。它会含有一个可null的变量并会在我们设置这个属性的时候分配一个真实的值。如果这个值在被获取之前没有被分配，它就会抛出一个异常。

这个在单例App这个例子中很有用：

```
class App : Application() {
    companion object {
        var instance: App by Delegates.notNull()
    }

    override fun onCreate() {
        super.onCreate()
        instance = this
    }
}
```

从Map中映射值

另外一种属性委托方式就是，属性的值会从一个map中获取value，属性的名字对应这个map中的key。这个委托可以让我们做一些很强大的事情，因为我们可以很简单地从一个动态地map中创建一个对象实例。如果我们import `kotlin.properties.getValue` ，我们可以从构造函数映射到 `val` 属性来得到一个不可修改的map。如果我们想去修改map和属性，我们也可以import `kotlin.properties.setValue` 。类需要一个 `MutableMap` 作为构造函数的参数。

想象我们从一个Json中加载了一个配置类，然后分配它们的key和value到一个map中。我们可以仅仅通过传入一个map的构造函数来创建一个实例：

```
import kotlin.properties.getValue

class Configuration(map: Map<String, Any?>) {
    val width: Int by map
    val height: Int by map
    val dp: Int by map
    val deviceName: String by map
}
```

作为一个参考，这里我展示下对于这个类怎么去创建一个必须要的map：

```
conf = Configuration(mapOf(
    "width" to 1080,
    "height" to 720,
    "dp" to 240,
    "deviceName" to "mydevice"
))
```

怎么去创建一个自定义的委托

先来说说我们要实现什么，举个例子，我们创建一个 `notNull` 的委托，它只能被赋值一次，如果第二次赋值，它就会抛异常。

Kotlin库提供了几个接口，我们自己的委托必须要实现： `ReadOnlyProperty` 和 `ReadWriteProperty` 。具体取决于我们被委托的对象是 `val` 还是 `var` 。

我们要做的第一件事就是创建一个类然后继承 `ReadWriteProperty` ：

```
private class NotNullSingleValueVar<T>() : ReadWriteProperty<Any
?, T> {

        override fun getValue(thisRef: Any?, property: KProperty
<*>): T {
            throw UnsupportedOperationException()
        }

        override fun setValue(thisRef: Any?, property: KProperty
<*>, value: T) {
        }
}
```

这个委托可以作用在任何非null的类型。它接收任何类型的引用，然后像getter和setter那样使用T。现在我们需要去实现这些函数。

- Getter函数 如果已经被初始化，则会返回一个值，否则会抛异常。
- Setter函数 如果仍然是null，则赋值，否则会抛异常。

```
private class NotNullSingleValueVar<T>() : ReadWriteProperty<Any
?, T> {
    private var value: T? = null
    override fun getValue(thisRef: Any?, property: KProperty<*>)
: T {
        return value ?: throw IllegalStateException("${desc.name
} " +
                "not initialized")
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>,
 value: T) {
        this.value = if (this.value == null) value
        else throw IllegalStateException("${desc.name} already i
nitialized")
    }
}
```

现在你可以创建一个对象，然后添加函数使用你的委托：

```
object DelegatesExt {
    fun notNullSingleValue<T>():
            ReadWriteProperty<Any?, T> = NotNullSingleValueVar()
}
```

重新实现Application单例化

在这个情景下，委托就可以帮助我们了。我们直到我们的单例不会是null，但是我们不能使用构造函数去初始化属性。所以我们可以使用 `notNull` 委托：

```
class App : Application() {
    companion object {
        var instance: App by Delegates.notNull()
    }

    override fun onCreate() {
            super.onCreate()
            instance = this
    }
}
```

这种情况下有个问题，我们可以在app的任何地方去修改这个值，因为如果我们使用 `Delegates.notNull()` ，属性必须是var的。但是我们可以使用刚刚创建的委托，这样可以多一点保护。我们只能修改这个值一次：

```
companion object {
    var instance: App by DelegatesExt.notNullSingleValue()
}
```

尽管，在这个例子中，使用单例可能是最简单的方法，但是我想用代码的形式展示给你怎么去创建一个自定义的委托。

创建一个SQLiteOpenHelper

如你所知，Android使用SQLite作为它的数据库管理系统。SQLite是一个嵌入app的一个数据库，它的确是非常轻量的。这就是为什么这是手机app的不错的选择。

尽管如此，它的操作数据库的API在Android中是非常原生的。你将会需要编写很多SQL语句和你的对象与 `ContentValues` 或者 `Cursors` 之间的解析过程。很感激的，联合使用Kotlin和Anko，我们可以大量简化这些。

当然，有很多Android中可以使用的关于数据库的库，多亏Kotlin的互操作性，所有这些库都可以正常使用。但是针对一个简单的数据库来说可以不使用任何它们，之后的一分钟之内你就可以看到。

ManagedSqliteOpenHelper

Anko提供了很多强大的SqliteOpenHelper来可以大量简化代码。当我们使用一个一般的 `SqliteOpenHelper` ，我们需要去调用 `getReadableDatabase()` 或者 `getWritableDatabase()` ，然后我们可以执行我们的搜索并拿到结果。在这之后，我们不能忘记调用 `close()` 。使用 `ManagedSqliteOpenHelper` 我们只需要：

```
forecastDbHelper.use {
    ...
}
```

在lambda里面，我们可以直接使用 `SqliteDatabase` 中的函数。它是怎么工作的？阅读Anko函数的实现方式真是一件有趣的事情，你可以从这里学到Kotlin的很多知识：

```
public fun <T> use(f: SQLiteDatabase.() -> T): T {
    try {
        return openDatabase().f()
    } finally {
        closeDatabase()
    }
}
```

首先， `use` 接收一个 `SQLiteDatabase` 的扩展函数。这表示，我们可以使用 `this` 在大括号中，并且处于 `SQLiteDatabase` 对象中。这个函数扩展可以返回一个值，所以我们可以像这么做：

```
val result = forecastDbHelper.use {
    val queriedObject = ...
    queriedObject
}
```

要记住，在一个函数中，最后一行表示返回值。因为T没有任何的限制，所以我们可以返回任何对象。甚至如果我们不想返回任何值就使用 `Unit` 。

由于使用了 `try-finally` ， `use` 方法会确保不管在数据库操作执行成功还是失败都会去关闭数据库。

而且，在 `sqliteDatabase` 中还有很多有用的扩展函数，我们会在之后使用到他们。但是现在让我们先定义我们的表和实现 `SqliteOpenHelper` 。

定义表

创建几个 `objects` 可以让我们避免表名列名拼写错误、重复等。我们需要两个表：一个用来保存城市的信息，另一个用来保存某天的天气预报。第二张表会有一个关联到第一张表的字段。

`CityForecastTable` 提供了表的名字还有需要列：一个id（这个城市的zipCode），城市的名称和所在国家。

```
object CityForecastTable {
    val NAME = "CityForecast"
    val ID = "_id"
    val CITY = "city"
    val COUNTRY = "country"
}
```

`DayForecast` 有更多的信息，就如你下面看到的有很多的列。最后一列 `cityId`，用来保持属于的城市id。

```
object DayForecastTable {
    val NAME = "DayForecast"
    val ID = "_id"
    val DATE = "date"
    val DESCRIPTION = "description"
    val HIGH = "high"
    val LOW = "low"
    val ICON_URL = "iconUrl"
    val CITY_ID = "cityId"
}
```

实现SqliteOpenHelper

我们 `SqliteOpenHelper` 的基本组成是数据库的创建和更新，并提供了一个 `SqliteDatebase` ，使得我们可以用它来工作。查询可以被抽取出来放在其它的类中：

```
class ForecastDbHelper() : ManagedSQLiteOpenHelper(App.instance,
        ForecastDbHelper.DB_NAME, null, ForecastDbHelper.DB_VERS
ION) {
    ...
}
```

我们在前面的章节中使用过我们创建的 `App.instance` ，这次我们同样的包括数据库名称和版本。这些值我们都会与 `SqliteOpenHelper` 一起定义在 `companion object` 中：

```
companion object {
    val DB_NAME = "forecast.db"
    val DB_VERSION = 1
    val instance: ForecastDbHelper by lazy { ForecastDbHelper()
}
}
```

`instance` 这个属性使用了 `lazy` 委托，它表示直到它真的被调用才会被创建。用这种方法，如果数据库从来没有被使用，我们没有必要去创建这个对象。一般 `lazy` 委托的代码块可以阻止在多个不同的线程中创建多个对象。这个只会发生在两个线程在同事时间访问这个 `instance` 对象，它很难发生但是发生具体还有看app的实现。无人如何， `lazy` 委托是线程安全的。

为了去创建这些定义的表，我们需要去提供一个 `onCreate` 函数的实现。当没有库使用的时候，创建表会通过我们编写原生的包含我们定义好的列和类型的 `CREATE TABLE` 语句来实现。然而Anko提供了一个简单的扩展函数，它接收一个表的名字和一系列由列名和类型构建的 `Pair` 对象：

```
db.createTable(CityForecastTable.NAME, true,
        Pair(CityForecastTable.ID, INTEGER + PRIMARY_KEY),
        Pair(CityForecastTable.CITY, TEXT),
        Pair(CityForecastTable.COUNTRY, TEXT))
```

- 第一个参数是表的名称
- 第二个参数，当是true的时候，创建之前会检查这个表是否存在。
- 第三个参数是一个 `Pair` 类型的 `vararg` 参数。 `vararg` 也存在在Java中，这是一种在一个函数中传入联系很多相同类型的参数。这个函数也接收一个对象数组。

Anko中有一种叫做 `SqlType` 的特殊类型，它可以与 `SqlTypeModifiers` 混合，比如 `PRIMARY_KEY` 。 `+` 操作符像之前那样被重写了。这个 `plus` 函数会把两者通过合适的方式结合起来，然后返回一个新的 `SqlType` ：

```
fun SqlType.plus(m: SqlTypeModifier) : SqlType {
    return SqlTypeImpl(name, if (modifier == null) m.toString()
            else "$modifier $m")
}
```

如你所见，她会把多个修饰符组合起来。

但是回到我们的代码，我们可以修改得更好。Kotlin标准库中包含了一个叫 `to` 的函数，又一次，让我们来展示Kotlin的强大之处。它作为第一参数的扩展函数，接收另外一个对象作为参数，把两者组装并返回一个 `Pair` 。

```
public fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

因为带有一个函数参数的函数可以被用于inline，所以结果非常清晰：

```
val pair = object1 to object2
```

然后，把他们应用到表的创建中：

```
db.createTable(CityForecastTable.NAME, true,
        CityForecastTable.ID to INTEGER + PRIMARY_KEY,
        CityForecastTable.CITY to TEXT,
        CityForecastTable.COUNTRY to TEXT)
```

这就是整个函数看起来的样子：

```
override fun onCreate(db: SQLiteDatabase) {
    db.createTable(CityForecastTable.NAME, true,
            CityForecastTable.ID to INTEGER + PRIMARY_KEY,
            CityForecastTable.CITY to TEXT,
            CityForecastTable.COUNTRY to TEXT)

    db.createTable(DayForecastTable.NAME, true,
            DayForecastTable.ID to INTEGER + PRIMARY_KEY + AUTOI
NCREMENT,
            DayForecastTable.DATE to INTEGER,
            DayForecastTable.DESCRIPTION to TEXT,
            DayForecastTable.HIGH to INTEGER,
            DayForecastTable.LOW to INTEGER,
            DayForecastTable.ICON_URL to TEXT,
            DayForecastTable.CITY_ID to INTEGER)
}
```

我们有一个相似的函数用于删除表。 `onUpgrade` 将只是删除表，然后重建它们。我们只是把我们数据库作为一个缓存，所以这是一个简单安全的方法保证我们的表会如我们所期望的那样被重建。如果我有很重要的数据需要保留，我们就需要优化 `onUpgrade` 的代码，让它根据数据库版本来做相应的数据转移。

```
override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newV
ersion: Int) {
    db.dropTable(CityForecastTable.NAME, true)
    db.dropTable(DayForecastTable.NAME, true)
    onCreate(db)
}
```

依赖注入

我试图不去增加很复杂的结构代码，保持简洁可测试性的代码和好的实践，我想我应该用Kotlin从其它方面去简化代码。如果你想了解一些控制反转或者依赖注入的话题，你可以查看我关于Android中使用Dagger注入的一系列文章。第一篇文章有关于他们这个团队的简单描写。

一种简单的方式，如果我们想拥有一些独立于其他类的类，这样更容易测试，并编写代码，易于扩展和维护，这时我们需要使用依赖注入。我们不去在类内部去实例化，我们在其它地方提供它们（通常通过构造函数）或者实例化它们。用这种方式，我们可以用其它的对象来替代他们。比如可以实现同样的接口或者在tests中使用mocks。

但是现在，这些依赖必须要在某些地方被提供，所以依赖注入由提供合作者的类组成。这些通常使用依赖注入器来完成。Dagger可能是Android上最流行的依赖注入器。当然当我们提供依赖有一定复杂性的时候是个很好的替代品。

但是最小的替代是可以在这个构造函数中使用默认值。我们可以给构造函数的参数通过分配默认值的方式提供一个依赖，然后在不同的情况下提供不同的实例。比如，在我们的 `ForecastDbHelper` 我们可以用更智能的方式提供一个context：

```
class ForecastDbHelper(ctx: Context = App.instance) :
        ManagedSQLiteOpenHelper(ctx, ForecastDbHelper.DB_NAME, null,
        ForecastDbHelper.DB_VERSION) {
        ...
}
```

现在我们有两种方式来创建这个类：

```
val dbHelper1 = ForecastDbHelper() // 它会使用 App.instance
val dbHelper2 = ForecastDbHelper(mockedContext) // 比如，提供给测试tests
```

我会到处使用这个特性，所以我在解释清楚之后再继续往下。我们已经有了表，所以是时候开始对它们增加和请求了。但是在这之前，我想先讲讲结合和函数操作符。别忘了查看代码库找到最新的代码。

集合和函数操作符

在我们这个项目我们已经使用过集合了，但是现在是时候展示它们结合函数操作符之后有多强大了。关于函数式编程很不错的一点是我们不用去解释我们怎么去做，而是直接说我想做什么。比如，如果我想去过滤一个list，不用去创建一个list，遍历这个list的每一项，然后如果满足一定的条件则放到一个新的集合中，而是直接食用filer函数并指明我想用的过滤器。用这种方式，我们可以节省大量的代码。

虽然我们可以直接用Java中的集合，但是Kotlin也提供了一些你希望用的本地的接口：

- **Iterable**：父类。所有我们可以遍历一系列的都是实现这个接口。
- **MutableIterable**：一个支持遍历的同时可以执行删除的Iterables。
- **Collection**：这个类相是一个范性集合。我们通过函数访问可以返回集合的size、是否为空、是否包含一个或者一些item。这个集合的所有方法提供查询，因为connections是不可修改的。
- **MutableCollection**：一个支持增加和删除item的Collection。它提供了额外的函数，比如`add`、`remove`、`clear`等等。
- **List**：可能是最流行的集合类型。它是一个范性有序的集合。因为它的有序，我们可以使用`get`函数通过position来访问。
- **MutableList**：一个支持增加和删除item的List。
- **Set**：一个无序并不支持重复item的集合。
- **MutableSet**：一个支持增加和删除item的Set。
- **Map**：一个key-value对的collection。key在map中是唯一的，也就是说不能有两对key是一样的键值对存在于一个map中。
- **MutableMap**：一个支持增加和删除item的map。

有很多不同集合可用的函数操作符。我想通过一个例子来展示给你看。知道有哪些可选的操作符是很有用的，因为这样会更容易分辨它们使用的时机。

总数操作符

any

如果至少有一个元素符合给出的判断条件，则返回true。

```
val list = listOf(1, 2, 3, 4, 5, 6)
assertTrue(list.any { it % 2 == 0 })
assertFalse(list.any { it > 10 })
```

all

如果全部的元素符合给出的判断条件，则返回true。

```
assertTrue(list.all { it < 10 })
assertFalse(list.all { it % 2 == 0 })
```

count

返回符合给出判断条件的元素总数。

```
assertEquals(3, list.count { it % 2 == 0 })
```

fold

在一个初始值的基础上从第一项到最后一项通过一个函数累计所有的元素。

```
assertEquals(25, list.fold(4) { total, next -> total + next })
```

foldRight

与 `fold` 一样，但是顺序是从最后一项到第一项。

```
assertEquals(25, list.foldRight(4) { total, next -> total + next
 })
```

forEach

遍历所有元素，并执行给定的操作。

```
list.forEach { println(it) }
```

forEachIndexed

与 `forEach` ，但是我们同时可以得到元素的index。

```
list.forEachIndexed { index, value
        -> println("position $index contains a $value") }
```

max

返回最大的一项，如果没有则返回null。

```
assertEquals(6, list.max())
```

maxBy

根据给定的函数返回最大的一项，如果没有则返回null。

```
// The element whose negative is greater
assertEquals(1, list.maxBy { -it })
```

min

返回最小的一项，如果没有则返回null。

```
assertEquals(1, list.min())
```

minBy

根据给定的函数返回最小的一项，如果没有则返回null。

```
// The element whose negative is smaller
assertEquals(6, list.minBy { -it })
```

none

如果没有任何元素与给定的函数匹配，则返回true。

```
// No elements are divisible by 7
assertTrue(list.none { it % 7 == 0 })
```

reduce

与 `fold` 一样，但是没有一个初始值。通过一个函数从第一项到最后一项进行累计。

```
assertEquals(21, list.reduce { total, next -> total + next })
```

reduceRight

与 `reduce` 一样，但是顺序是从最后一项到第一项。

```
assertEquals(21, list.reduceRight { total, next -> total + next
})
```

sumBy

返回所有每一项通过函数转换之后的数据的总和。

```
assertEquals(3, list.sumBy { it % 2 })
```

过滤操作符

drop

返回包含去掉前n个元素的所有元素的列表。

```
assertEquals(listOf(5, 6), list.drop(4))
```

dropWhile

返回根据给定函数从第一项开始去掉指定元素的列表。

```
assertEquals(listOf(3, 4, 5, 6), list.dropWhile { it < 3 })
```

dropLastWhile

返回根据给定函数从最后一项开始去掉指定元素的列表。

```
assertEquals(listOf(1, 2, 3, 4), list.dropLastWhile { it > 4 })
```

filter

过滤所有符合给定函数条件的元素。

```
assertEquals(listOf(2, 4, 6), list.filter { it % 2 == 0 })
```

filterNot

过滤所有不符合给定函数条件的元素。

```
assertEquals(listOf(1, 3, 5), list.filterNot { it % 2 == 0 })
```

filterNotNull

过滤所有元素中不是null的元素。

```
assertEquals(listOf(1, 2, 3, 4), listWithNull.filterNotNull())
```

slice

过滤一个list中指定index的元素。

```
assertEquals(listOf(2, 4, 5), list.slice(listOf(1, 3, 4)))
```

take

返回从第一个开始的n个元素。

```
assertEquals(listOf(1, 2), list.take(2))
```

takeLast

返回从最后一个开始的n个元素

```
assertEquals(listOf(5, 6), list.takeLast(2))
```

takeWhile

返回从第一个开始符合给定函数条件的元素。

```
assertEquals(listOf(1, 2), list.takeWhile { it < 3 })
```

映射操作符

flatMap

遍历所有的元素，为每一个创建一个集合，最后把所有的集合放在一个集合中。

```
assertEquals(listOf(1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7),
list.flatMap { listOf(it, it + 1) })
```

groupBy

返回一个根据给定函数分组后的map。

```
assertEquals(mapOf("odd" to listOf(1, 3, 5), "even" to listOf(2,
4, 6)), list.groupBy { if (it % 2 == 0) "even" else "odd" })
```

map

返回一个每一个元素根据给定的函数转换所组成的List。

```
assertEquals(listOf(2, 4, 6, 8, 10, 12), list.map { it * 2 })
```

mapIndexed

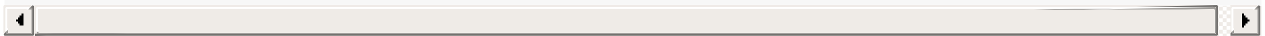
返回一个每一个元素根据给定的包含元素index的函数转换所组成的List。

```
assertEquals(listOf (0, 2, 6, 12, 20, 30), list.mapIndexed { ind
ex, it -> index * it })
```

mapNotNull

返回一个每一个非null元素根据给定的函数转换所组成的List。

```
assertEquals(listOf(2, 4, 6, 8), listWithNull.mapNotNull { it * 2
 })
```



元素操作符

contains

如果指定元素可以在集合中找到，则返回true。

```
assertTrue(list.contains(2))
```

elementAt

返回给定index对应的元素，如果index数组越界则会抛出 `IndexOutOfBoundsException` 。

```
assertEquals(2, list.elementAt(1))
```

elementAtOrElse

返回给定index对应的元素，如果index数组越界则会根据给定函数返回默认值。

```
assertEquals(20, list.elementAtOrElse(10, { 2 * it }))
```

elementAtOrNull

返回给定index对应的元素，如果index数组越界则会返回null。

```
assertNull(list.elementAtOrNull(10))
```

first

返回符合给定函数条件的第一个元素。

```
assertEquals(2, list.first { it % 2 == 0 })
```

firstOrNull

返回符合给定函数条件的第一个元素，如果没有符合则返回null。

```
assertNull(list.firstOrNull { it % 7 == 0 })
```

indexOf

返回指定元素的第一个index，如果不存在，则返回 `-1` 。

```
assertEquals(3, list.indexOf(4))
```

indexOfFirst

返回第一个符合给定函数条件的元素的index，如果没有符合则返回 `-1` 。

```
assertEquals(1, list.indexOfFirst { it % 2 == 0 })
```

indexOfLast

返回最后一个符合给定函数条件的元素的index，如果没有符合则返回 `-1` 。

```
assertEquals(5, list.indexOfLast { it % 2 == 0 })
```

last

返回符合给定函数条件的最后一个元素。

```
assertEquals(6, list.last { it % 2 == 0 })
```

lastIndexOf

返回指定元素的最后一个index，如果不存在，则返回 `-1` 。

lastOrNull

返回符合给定函数条件的最后一个元素，如果没有符合则返回null。

```
val list = listOf(1, 2, 3, 4, 5, 6)
assertNull(list.lastOrNull { it % 7 == 0 })
```

single

返回符合给定函数的单个元素，如果没有符合或者超过一个，则抛出异常。

```
assertEquals(5, list.single { it % 5 == 0 })
```

singleOrNull

返回符合给定函数的单个元素，如果没有符合或者超过一个，则返回null。

```
assertNull(list.singleOrNull { it % 7 == 0 })
```

生产操作符

merge

把两个集合合并成一个新的，相同index的元素通过给定的函数进行合并成新的元素作为新的集合的一个元素，返回这个新的集合。新的集合的大小由最小的那个集合大小决定。

```
val list = listOf(1, 2, 3, 4, 5, 6)
val listRepeated = listOf(2, 2, 3, 4, 5, 5, 6)
assertEquals(listOf(3, 4, 6, 8, 10, 11), list.merge(listRepeated
) { it1, it2 -> it1 + it2 })
```

partition

把一个给定的集合分割成两个，第一个集合是由原集合每一项元素匹配给定函数条件返回 `true` 的元素组成，第二个集合是由原集合每一项元素匹配给定函数条件返回 `false` 的元素组成。

```
assertEquals(
    Pair(listOf(2, 4, 6), listOf(1, 3, 5)),
    list.partition { it % 2 == 0 }
)
```

plus

返回一个包含原集合和给定集合中所有元素的集合，因为函数的名字原因，我们可以使用 `+` 操作符。

```
assertEquals(
    listOf(1, 2, 3, 4, 5, 6, 7, 8),
    list + listOf(7, 8)
)
```

zip

返回由 `pair` 组成的List，每个 `pair` 由两个集合中相同index的元素组成。这个返回的List的大小由最小的那个集合决定。

```
assertEquals(
    listOf(Pair(1, 7), Pair(2, 8)),
    list.zip(listOf(7, 8))
)
```

unzip

从包含pair的List中生成包含List的Pair。

```
assertEquals(
    Pair(listOf(5, 6), listOf(7, 8)),
    listOf(Pair(5, 7), Pair(6, 8)).unzip()
)
```

顺序操作符

reverse

返回一个与指定list相反顺序的list。

```
val unsortedList = listOf(3, 2, 7, 5)
assertEquals(listOf(5, 7, 2, 3), unsortedList.reverse())
```

sort

返回一个自然排序后的list。

```
assertEquals(listOf(2, 3, 5, 7), unsortedList.sort())
```

sortBy

返回一个根据指定函数排序后的list。

```
assertEquals(listOf(3, 7, 2, 5), unsortedList.sortBy { it % 3 })
```

sortDescending

返回一个降序排序后的List。

```
assertEquals(listOf(7, 5, 3, 2), unsortedList.sortDescending())
```

sortDescendingBy

返回一个根据指定函数降序排序后的list。

```
assertEquals(listOf(2, 5, 7, 3), unsortedList.sortDescendingBy {
 it % 3 })
```

从数据库中保存或查询数据

前面一个章节中我们讲了关于`SQLiteOpenHelper`的创建，但是我们需要在必要的时候有方法去保存我们的数据到数据库，或者从我们的数据库中查询数据。另外一个叫`ForecastDb`类就会做这件事。

创建数据库model类

但是首先，我们要去为数据库创建model类。你还记得我们之前所见的map委托的方式？我们要把这些属性直接映射到数据库中，反过来也一样。

我们先来看下 `CityForecast` 类：

```
class CityForecast(val map: MutableMap<String, Any?>,
                   val dailyForecast: List<DayForecast>)
 {
    var _id: Long by map
    var city: String by map
    var country: String by map

    constructor(id: Long, city: String, country: String,
                dailyForecast: List<DayForecast>)
    : this(HashMap(), dailyForecast) {
        this._id = id
        this.city = city
        this.country = country
    }
}
```

默认的构造函数会得到一个含有属性和对应的值的map，和一个dailyForecast。多亏了委托，这些值会根据key的名字会映射到相应的属性中去。如果我们希望映射的过程运行完美，那么属性的名字必须要和数据库中对应的名字一模一样。我们后面会讲原因。

但是，第二个构造函数也是必要的。这是因为我们需要从domain映射到数据库类中，所以不能使用map，从属性中设置值也是方便的。我们传入一个空的map，但是又一次，多亏了委托，当我们设置值到属性的时候，它会自动增加所有的值到map中。用这种方式，我们就准备好map来保存到数据库中了。使用了这些有用的代码，我将会看见它运行起来就像魔法一样神奇。

现在我们需要第二个类，DayForecast，它会是第二个表。它包括表中的每一列作为它的属性，它也有第二个构造函数。唯一不同之处就是不需要设置id，因为它将通过SQLite自增长。

```
class DayForecast(var map: MutableMap<String, Any?>) {
    var _id: Long by map
    var date: Long by map
    var description: String by map
    var high: Int by map
    var low: Int by map
    var iconUrl: String by map
    var cityId: Long by map

    constructor(date: Long, description: String, high: Int, low: Int,
            iconUrl: String, cityId: Long)
    : this(HashMap()) {
        this.date = date
        this.description = description
        this.high = high
        this.low = low
        this.iconUrl = iconUrl
        this.cityId = cityId
    }
}
```

这些类将会帮助我们SQLite表与对象之间的互相映射。

写入和查询数据库

`SqliteOpenHelper` 只是一个工具，是SQL世界和OOP之间的一个通道。我们要新建几个类来请求已经保存在数据库中的数据，和保存新的数据。被定义的类会使用 `ForecastDbHelper` 和 `DataMapper` 来转换数据库中的数据到 `domain models` 。我仍旧使用默认值的方式来实现简单的依赖注入：

```
class ForecastDb(
    val forecastDbHelper: ForecastDbHelper = ForecastDbHelper.in
stance,
    val dataMapper: DbDataMapper = DbDataMapper()) {
    ...
}
```

所有的函数使用前面章节讲到过的 `use()` 函数。lambda返回的值也会被作为这个函数的返回值。所以让我们定义一个使用 `zip code` 和 `date` 来查询一个 `forecast` 的函数：

```
fun requestForecastByZipCode(zipCode: Long, date: Long) = foreca
stDbHelper.use {
    ...
}
```

这么没有什么解释的：我们使用 `use` 函数返回的结果作为这个函数返回的结果。

查询一个forecast

第一个要做的查询就是每日的天气预报，因为我们需要这个列表来创建一个 `city` 对象。Anko提供了一个简单的请求构建器，所以我们来利用下这个有利条件：

```
val dailyRequest = "${DayForecastTable.CITY_ID} = ? " +
    "AND ${DayForecastTable.DATE} >= ?"

val dailyForecast = select(DayForecastTable.NAME)
        .whereSimple(dailyRequest, zipCode.toString(), date.toSt
ring())
        .parseList { DayForecast(HashMap(it)) }
```

第一行， `dailyRequest` 是查询语句中 `where` 的一部分。它是 `whereSimple` 函数需要的第一个参数，这与我们用一般的helper做的方式很相似。这里有另外一个简化的 `where` 函数，它需要一些tags和values来进行匹配。我不太喜欢这个方式，因为我觉得这个增加了代码的模版化，虽然这个对我们把values解析成String很有利。最后它看起来会是这样：

```
val dailyRequest = "${DayForecastTable.CITY_ID} = {id}" + "AND $
{DayForecastTable.DATE} >= {date}"

val dailyForecast = select(DayForecastTable.NAME)
        .where(dailyRequest, "id" to zipCode, "date" to date)
        .parseList { DayForecast(HashMap(it)) }
```

你可以选择你喜欢的一种方式。 `select` 函数是很简单的，它仅仅是需要一个被查询表的名字。 `parse` 函数的时候会有一些魔法在里面。在这个例子中我们假设请求结果是一个list，使用了 `parseList` 函数。它使用了 `RowParser` 或 `RapRowParser` 函数去把cursor转换成一个对象的集合。这两个不同之处就是 `RowParser` 是依赖列的顺序的，而 `MapRowParser` 是从map中拿到作为column的key名的。

在它们之间有两个重载的冲突，所以我们不能直接使用简化的方式准确地创建需要的对象。但是没有什么是不能通过扩展函数来解决的。我创建了一个接收一个lambda函数返回一个 `MapRowParser` 的函数。解析器会调用这个lambda来创建这个对象：

```
fun <T : Any> SelectQueryBuilder.parseList(
    parser: (Map<String, Any>) -> T): List<T> =
        parseList(object : MapRowParser<T> {
            override fun parseRow(columns: Map<String, Any>): T
= parser(columns)
})
```

这个函数可以帮助我们简单地去 `parseList` 查询的结果：

```
parseList { DayForecast(HashMap(it)) }
```

解析器接收的 `immutable map` 被我们转化成了一个 `mutable map` （我们需要在 `database model` 中是可以修改的）通过使用相应的 `HashMap` 构造函数。在 `DayForecast` 中的构造函数中会使用到这个 `HashMap` 。

所以，这个查询返回了一个 `Cursor` ，要理解这个场景的背后到底发生了什么。 `parseList` 中会迭代它，然后得到 `Cursor` 的每一行直到最后一个。对于每一行，它会创建一个包含这列的key和给对应的key赋值后的map。然后把这个map返回给这个解析器。

如果查询没有任何结果， `parseList` 会返回一个空的list。

下一步查询城市也是一样的方法：

```
val city = select(CityForecastTable.NAME)
        .whereSimple("${CityForecastTable.ID} = ?", zipCode.toSt
ring())
        .parseOpt { CityForecast(HashMap(it), dailyForecast) }
```

不同之处是：我们使用的是 `parseOpt` 。这个函数返回一个可null的对象。结果可以使一个null或者单个的对象，这取决于请求是否能在数据库中查询到数据。这里有另外一个叫 `parseSingle` 的函数，本质上是一样的，但是它返回的事一个不可null的对象。所以如果没有在数据库中找到这一条数据，它会抛出一个异常。在我们的例子中，第一次查询一个城市的时候，肯定是不存在的，所以使用 `parseOpt` 会更安全。我又创建了一个好用的函数来阻止我们需要的对象的创建：

```
public fun <T : Any> SelectQueryBuilder.parseOpt(
    parser: (Map<String, Any>) -> T): T? =
        parseOpt(object : MapRowParser<T> {
            override fun parseRow(columns: Map<String, Any>): T
= parser(columns)
        })
```

最后如果返回的city不是null，我们使用 `dataMapper` 把它转换成 `domain object` 再返回它。否则，我们直接返回null。你应该记得，lambda的最后一行表示返回值。所以这里将会返回一个 `CityForecast?` 类型的对象：

```
if (city != null) dataMapper.convertToDomain(city) else null
```

`DataMapper` 函数很简单：

```
fun convertToDomain(forecast: CityForecast) = with(forecast) {
    val daily = dailyForecast.map { convertDayToDomain(it) }
    ForecastList(_id, city, country, daily)
}

private fun convertDayToDomain(dayForecast: DayForecast) = with(
dayForecast) {
    Forecast(date, description, high, low, iconUrl)
}
```

最后完整的函数如下：

```
fun requestForecastByZipCode(zipCode: Long, date: Long) = foreca
stDbHelper.use {

    val dailyRequest = "${DayForecastTable.CITY_ID} = ? AND " +
        "${DayForecastTable.DATE} >= ?"
    val dailyForecast = select(DayForecastTable.NAME)
            .whereSimple(dailyRequest, zipCode.toString(), date.
toString())
            .parseList { DayForecast(HashMap(it)) }

    val city = select(CityForecastTable.NAME)
            .whereSimple("${CityForecastTable.ID} = ?", zipCode.
toString())
            .parseOpt { CityForecast(HashMap(it), dailyForecast)
 }

    if (city != null) dataMapper.convertToDomain(city) else null
}
```

另外一个Anko中好玩的功能我们在这里展示，那就是你可以使用 `classParser()` 来替代我们用的 `MapRowParser` ，它是基于列名通过反射的方式去生成对象的。我喜欢另一种方法因为我不需要使用反射并且还有控制权进行转换操作，但是在有时候可能对你有用。

保存一个forecast

`saveForecast` 函数只是从数据库中清除数据，然后转换 `domain` model为数据库model，然后插入每一天的 `forecast` 和 `city forecast` 。这个结构比之前的更简单：它通过 `use` 函数从 `database helper` 中返回数据。在这个例子中我们不需要返回值，所以它将返回 `Unit` 。

```
fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {

    ...
}
```

首先，我们清空这两个表。Anko没有提供比较漂亮的方式来做这个，但这并不意味着我们不行。所以我们创建了一个 `SQLiteDatabase` 的扩展函数来让我们可以像SQL查询一样来执行它：

```
fun SQLiteDatabase.clear(tableName: String) {
    execSQL("delete from $tableName")
}
```

清空这两个表：

```
clear(CityForecastTable.NAME)
clear(DayForecastTable.NAME)
```

现在，是时候去转换执行 `insert` 后返回的数据了。在这一点上你可能直到我是 `with` 函数的粉丝：

```
with(dataMapper.convertFromDomain(forecast)) {
    ...
}
```

从 `domain model` 转换的方式也是很直接的：

```
fun convertFromDomain(forecast: ForecastList) = with(forecast) {
    val daily = dailyForecast.map { convertDayFromDomain(id, it)
 }
    CityForecast(id, city, country, daily)
}

private fun convertDayFromDomain(cityId: Long, forecast: Forecast
) =
    with(forecast) {
        DayForecast(date, description, high, low, iconUrl, cityI
d)
    }
```

在代码块，我们可以在不使用引用和变量的情况下使用 `dailyForecast` 和 `map` ，只是像我们在这个类内部一样就可以了。针对插入我们使用另外一个Anko函数，它需要一个表名和一个 `vararg` 修饰的 `Pair<String, Any>` 作为参数。这个函数会把 `vararg` 转换成Android SDK需要的 `ContentValues` 对象。所以我们的任务组成是把 `map` 转换成一个 `vararg` 数组。我们为 `MutableMap` 创建了一个扩展函数：

```
fun <K, V : Any> MutableMap<K, V?>.toVarargArray():
    Array<out Pair<K, V>> =  map({ Pair(it.key, it.value!!) }).t
oTypedArray()
```

它是支持可null的值的（这是 `map delegate` 的条件），把它转换为非null值（ `select` 函数需要）的 `Array` 所组成的 `Pairs` 。不用担心就算你不完全理解这个函数，我很快就会讲到可空性。

所以，这个新的函数我们可以这么使用：

```
insert(CityForecastTable.NAME, *map.toVarargArray())
```

它在 `CityForecast` 中插入了一个一行新的数据。在 `toVarargArray` 函数结果前面使用 `*` 表示这个array会被分解成为一个 `vararg` 参数。这个在Java中是自动处理的，但是我们需要在Kotlin中明确指明。

每天的天气预报也是一样了：

```
dailyForecast.forEach { insert(DayForecastTable.NAME, *it.map.to
VarargArray()) }
```

所以，通过 `map` 的使用，我们可以用很简单的方式把类转换为数据表，反之亦然。因为我们已经新建了扩展函数，我们可以在别的项目中使用，这个才是真正可贵的地方。

这个函数的完整代码如下：

```
fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {

    clear(CityForecastTable.NAME)
    clear(DayForecastTable.NAME)

    with(dataMapper.convertFromDomain(forecast)) {
        insert(CityForecastTable.NAME, *map.toVarargArray())
        dailyForecast forEach {
            insert(DayForecastTable.NAME, *it.map.toVarargArray(
))
        }
    }
}
```

在这一章中有很多代码被需要，所以你可以到代码库中查看检出。

Kotlin中的null安全

如果你正在使用Java 7工作的话，null安全是Kotlin中最令人感兴趣的特性之一了。但是就如你在本书中看到的，它好像不存在一样，一直到上一章我们几乎都不需要去担心它。

通过我们自己创造的亿万美金的错误对null的思考，我们有时候的确需要去定义一个变量包不包含一个值。在Java中尽管注解和IDE在这方面帮了我们很多，但是我们仍然可以这么做：

```
Forecast forecast = null;
forecast.toString();
```

这个代码可以被完美地编译（你可能会从IDE上得到一个警告），然后正常地执行，但是显然它会抛一个 `NullPointerException` 。这个相当不安全的。而且按照我们的想法，我们应该去控制一切，随着代码的增长，我们会慢慢对某些null的控制。所以最终会得到很多的 `NullPointerException` 或者丢失很多null检查（可能两者混合）。

可null类型怎么工作

大部分现代语言使用某些方法去解决了这个问题，Kotlin的方法跟别的相似的语言比是相当另类和不同的。但是黄金准则还是一样：如果变量是可以是null，编译器强制我们去用某种方式去处理。

指定一个变量是可null是通过在类型的最后增加一个问号。因为在Kotlin中一切都是对象（甚至是Java中原始数据类型），一切都是可null的。所以，当然我们可以有一个可null的integer：

```
val a: Int? = null
```

一个可nul类型，你在没有进行检查之前你是不能直接使用它。这个代码不能被编译：

```
val a: Int? = null
a.toString()
```

前一行代码标记为可null，然后编译器就会知道它，所以在你null检查之前你不能去使用它。还有一个特性是当我们检查了一个对象的可null性，之后这个对象就会自动转型成不可null类型，这就是Kotlin编译器的智能转换：

```
vala:Int?=null
...
if(a!=null){
    a.toString()
}
```

在 `if` 中， `a` 从 `Int?` 变成了 `Int` ，所以我们可以不需要再检查可null性而直接使用它。 `if` 代码之外，当然我们又得检查处理。这仅仅在变量当前不能被改变的时候才有效，因为否则这个value可能被另外的线程修改，这时前面的检查会返回false。 `val` 属性或者本地（ `val or var` ）变量。

这听起来会让事情变得更多。难道我们不得不去编写大量代码去进行可null性的检查？当然不是，首先，因为大多数时候你不需要使用null类型。Null引用没有我们想象中的有用，当你想弄清楚一个变量是否可以为null时你就会发现这一点。但是Kotlin也有它自己的使处理更简洁的方案。举个例子，我们如下简化代码：

```
val a: Int? = null
...
a?.toString()
```

这里我们使用了安全访问操作符(`?`)。只有这个变量不是null的时候才会去执行前面的那行代码。否则，它不会做任何事情。并且我们甚至可以使用**Elvis operator**(`?:`)：

```
val a:Int? = null
val myString = a?.toString() ?: ""
```

因为在Kotlin中 `throw` 和 `return` 都是表达式，他们可以用在**Elvis operator**操作符的右边：

```
val myString = a?.toString() ?: return false

val myString = a?.toString() ?: throw IllegalStateException()
```

然后，我们可能会遇到这种情景，我们确定我们是在用一个非null变量，但是他的类型却是可null的。我们可以使用 `!!` 操作符来强制编译器执行可null类型时跳过限制检查：

```
val a: Int? = null
a!!.toString()
```

上面的代码将会被编译，但是很显然会奔溃。所以我们要确保只能在特定的情况下使用。通常我们可以自己选择作为解决方案。如果一份代码满篇都是 `!!` ，那就有股代码没有被正确处理的气味了。

可null性和Java库

好了，前面的章节解释了使用Kotlin代码完美地工作。但是与普通的Java库和Android SDK会发生什么呢？在Java中，所有对象可以被定义为null。所以我们不得不处理大量潜在的在现实中不可能是null的null变量。这意味着我们的代码最后可能会有几百个 `!!` 操作符，这绝对不是一个好的主意。

当我们去处理Android SDK时，你可能看见所有Java方法的参数被标记为单个的 `!` 。比如，Java中在一些获取对象的方法在Kotlin中显示返回 `Any!` 。这表示让开发者自己决定是否这个变量是否可null。

很幸运，新版本的Android开始使用 `@Nullable` 和 `@NonNull` 注解来辨别参数是否可以是null或者否个函数是否可以返回null。当我们怀疑时，我们可以进入源码去检查是否会接收到一个null对象。我的猜想是在以后，编译器能够读取这些注解，然后强制（或者至少是建议）一个更好的方法。

现在开始，当一个Jetbrains的 `@Nullable` 注解（这个与Android的注解不同）被注解在一个非null的变量时，就会获得一个警告。相对的没有发生在 `@NotNull` 注解上。

所以我们来举个例子，如果我们创建了一个Java的测试类：

```
import org.jetbrains.annotations.Nullable;
public class NullTest {

    @Nullable
    public Object getObject(){
        return "";
    }
}
```

然后在Kotlin中使用：

```
val test = NullTest()
val myObject: Any = test.getObject()
```

我们会发现，在 `getObject` 函数上会显示一个警告。但是这只是从现在才开始的编译器检查，并且它还不认识Android的注解，所以我们可能不得不花更多的时间来等待一个更智能的方式。不管怎么样，使用源码注解的方式和一些Androd SDK的知识，我们也很难犯错误。

比如重写 `Activity` 的 `onCraete` 函数，我们可以决定是否让 `savedInstanceState` 可null：

```
override fun onCreate(savedInstanceState: Bundle?) {
}

override fun onCreate(savedInstanceState: Bundle) {
}
```

这两种方法都会被编译，但是第二种是错误的，因为一个Activity很可能接收到一个null的bundle。只要小心一点点就足够了。当你有疑问时，你可以就用可null的对象然后处理掉用可能的null。记住，如果你使用了 `!!` ，可能是因为你确信对象不可能为null，如果是这样，请定义为非null。

这个灵活性在Java库中真的很有必要，而且随着编译器的进化，我们将可能看到更好的交互（可能是基于注解的），但是现在来说这个机制已经足够灵活了。

创建业务逻辑来访问数据

在实现访问服务器和与本地数据库交互之后，是时候把事情整合起来了。逻辑步骤如下：

- 从数据库获取数据
- 检查是否存在对应星期的数据
- 如果有，返回UI并且渲染
- 如果没有，请求服务器获取数据
- 结果被保存在数据库中并且返回UI渲染

但是我们的 `commands` 不应该去处理所有这些逻辑。数据源应该是一个具体的实现，这样就可以被容易地修改，所以增加一些额外的代码，然后把 `command` 从数据访问中抽象出来听起来是个不错的方式。在我们的实现中，它会遍历整个list直到结果被找到。

所以我们先来给接口定义一些我们实现 `provider` 需要使用到的数据源：

```
interface ForecastDataSource {
    fun requestForecastByZipCode(zipCode: Long, date: Long): For
ecastList?
}
```

`provider` 需要一个接收 `zip code` 和一个 `date` ，然后它应该根据那一天返回一周的天气预报。

```
class ForecastProvider(val sources: List<ForecastDataSource> =
        ForecastProvider.SOURCES) {

    companion object {
        val DAY_IN_MILLIS = 1000 * 60 * 60 * 24
        val SOURCES = listOf(ForecastDb(), ForecastServer())
    }
    ...
}
```

`forecast provider` 接收一个数据源列表，通过构造函数传入（比如用于测试），但是我设置了source的默认值为被定义在 `companion object` 中的 `SOURCES` List。我将使用数据库的数据源和服务端数据源。顺序是很重要的，因为它会根据顺序去遍历这个sources，然后一旦获取到有效的返回值就会停止查询。逻辑顺序是先在本地查询（本地数据库中），然后再通过API查询。

所以主函数的代码如下：

```
fun requestByZipCode(zipCode: Long, days: Int): ForecastList
            = sources.firstResult { requestSource(it, days, zipC
ode) }
```

它会得到第一个不是null的结果然后返回。当我在第18章中讲到的大量的函数操作符中搜索后，我没有找到完全符合我想要的。所以当我去查看Kotlin的源码时，我直接拷贝了 `first` 函数然后修改它们来达到我想要的目的：

```
inline fun <T, R : Any> Iterable<T>.firstResult(predicate: (T) -
> R?) : R {
    for (element in this){
        val result = predicate(element)
        if (result != null) return result
    }
    throw NoSuchElementException("No element matching predicate
was found.")
}
```

该函数接收一个断言函数，它接收一个 `T` 类型的对象然后返回一个 `R?` 类型的值。这表示 `predicate` 可以返回null类型，但是我们的 `firstResult` 不能返回null。这就是为什么返回 `R` 的原因。

它怎么工作呢？它将遍历集合中的每一个元素然后执行这个断言函数。当这个断言函数的结果返回不是null时，这个结果就会被返回。

如果我们可以允许sources返回null，那我们就可以使用 `firstOrNull` 函数来代替。不同之处就是最后一行的返回null和抛异常。但是我现在不在代码里面去处理这些细节了。

在我们的例子中 `T = ForecastDataSource` ， `R = ForecastList` 。但是记住在 `ForecastDataSource` 中指定的函数返回一个 `ForecastList?` ，也就是 `R?` ，所以一切都是匹配得这么完美。 `requestSource` 让前面的函数看起来更有可读性：

```
fun requestSource(source: ForecastDataSource, days: Int, zipCode
: Long):
        ForecastList? {
    val res = source.requestForecastByZipCode(zipCode, todayTime
Span())
    return if (res != null && res.size() >= days) res else null
}
```

如果结果不是null并且数量也参数匹配，那这个查询被执行且只会返回一个数据。否则，数据源没有足够的数据来返回一个成功的结果。

函数 `todayTimeSpan()` 计算今天毫秒级的时间，并排除掉“时差”。其中一些数据源（我们例子中的数据库）可能会需要它。因为如果我们没有指定更多的信息，服务端默认就是今天，所以我们不需要设置它。

```
private fun todayTimeSpan() = System.currentTimeMillis() / DAY_I
N_MILLIS * DAY_IN_MILLIS
```

这个类完整的代码如下：

```
class ForecastProvider(val sources: List<ForecastDataSource> =
        ForecastProvider.SOURCES) {

    companion object {
        val DAY_IN_MILLIS = 1000 * 60 * 60 * 24;
        val SOURCES = listOf(ForecastDb(), ForecastServer())
    }

    fun requestByZipCode(zipCode: Long, days: Int): ForecastList
            = sources.firstResult { requestSource(it, days, zipC
ode) }

    private fun requestSource(source: RepositorySource, days: In
t,
            zipCode: Long): ForecastList? {
        val res = source.requestForecastByZipCode(zipCode, today
TimeSpan())
        return if (res != null && res.size() >= days) res else n
ull
    }

    private fun todayTimeSpan() = System.currentTimeMillis() /
            DAY_IN_MILLIS * DAY_IN_MILLIS
}
```

我们已经定义了一个 `ForecastDb` 。现在我们需要去实现 `ForcastDataSource` ：

```
class ForecastDb(val forecastDbHelper: ForecastDbHelper =
        ForecastDbHelper.instance, val dataMapper: DbDataMapper
= DbDataMapper())
        : ForecastDataSource {

    override fun requestForecastByZipCode(zipCode: Long, date: L
ong) =
            forecastDbHelper.use {
            ...
    }
    ...
}
```

`ForecastServer` 还没有还被实现，但是这是非常简单的。它在从服务端接收到数据之后就会使用 `ForecastDb` 去保存到数据库。用这种方式，我们就可以缓存这些数据到数据库中，提供给以后的查询。

```
class ForecastServer(val dataMapper: ServerDataMapper = ServerDa
taMapper(),
        val forecastDb: ForecastDb = ForecastDb()) : ForecastDat
aSource {

    override fun requestForecastByZipCode(zipCode: Long, date: L
ong):
            ForecastList? {
        val result = ForecastByZipCodeRequest(zipCode).execute()
        val converted = dataMapper.convertToDomain(zipCode, resu
lt)
        forecastDb.saveForecast(converted)
        return forecastDb.requestForecastByZipCode(zipCode, date
)
    }
}
```

它也是使用了之前我们创建的 `data mapper` ，最然我们修改一些函数的名字来让它更加与我们之前用在 `database model` 的mapper更相似。你可以查看 `provider` 来查看细节。

被重写的方法用来请求服务器，转换结果到 `domain objects` 并保存它们到数据库。它最后查询数据库返回数据，这是因为我们需要使用到插入到数据库中的字增长id。

这就是 `provider` 被实现的最后的一步了。现在我们需要开始使用它。 `ForecastCommand` 不会再直接与服务端交互，也不会转换数据到 `domain model` 。

```
RequestForecastCommand(val zipCode: Long,
        val forecastProvider: ForecastProvider = ForecastProvide
r()) :
        Command<ForecastList> {

    companion object {
        val DAYS = 7
    }

    override fun execute(): ForecastList {
        return forecastProvider.requestByZipCode(zipCode, DAYS)
    }
}
```

其它修改的地方包括重命名和包的结构调整。在Kotlin for Android Developers repository查看相应的提交。

Flow control和ranges

我在我们的代码中使用了一些条件表达式，但是现在是时候去更深地去解释它们了。我们通常都在使用过程式编程语言的时候很少地去使用代码流控制的机制去编写（有些过程式编程语言中几乎已消失），但是它们还是很有用的。这也是一个新的强大的想法让解决一些特定的情况下的问题变得更容易。

If表达式

在Kotlin中一切都是表达式，也就是说一切都返回一个值。如果 `if` 条件不含有一个exception，那我们可以像我们平时那样使用它：

```
if(x>0){
    toast("x is greater than 0")
}else if(x==0){
    toast("x equals 0")
}else{
    toast("x is smaller than 0")
}
```

我们也可以把结果赋值给一个变量。我们在我们的代码中使用了很多次：

```
val res = if (x != null && x.size() >= days) x else null
```

这也说明我也不需要像Java那种有一个三元操作符，因为我们可以使用它来简单实现：

```
val z = if (condition) x else y
```

所以 `if` 表达式总是返回一个value。如果一个分支返回了Unit，那整个表达式也将返回Unit，它是可以被忽略的，这种情况下它的用法也就跟一般Java中的 `if` 条件一样了。

When表达式

`when` 表达式与Java中的 `switch/case` 类似，但是要强大得多。这个表达式会去试图匹配所有可能的分支直到找到满意的一项。然后它会运行右边的表达式。与Java的 `switch/case` 不同之处是参数可以是任何类型，并且分支也可以是一个条件。

对于默认的选项，我们可以增加一个 `else` 分支，它会在前面没有任何条件匹配时再执行。条件匹配成功后执行的代码也可以是代码块：

```
when (x){
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("I'm a block")
        print("x is neither 1 nor 2")
    }
}
```

因为它是一个表达式，它也可以返回一个值。我们需要考虑什么时候作为一个表达式使用，它必须要覆盖所有分支的可能性或者实现 `else` 分支。否则它不会被编译成功：

```
val result = when (x) {
    0, 1 -> "binary"
    else -> "error"
}
```

如你所见，条件可以是一系列被逗号分割的值。但是它可以更多的匹配方式。比如，我们可以检测参数类型并进行判断：

```
when(view) {
    is TextView -> view.setText("I'm a TextView")
    is EditText -> toast("EditText value: ${view.getText()}")
    is ViewGroup -> toast("Number of children: ${view.getChildCo
unt()} ")
    else -> view.visibility = View.GONE
}
```

再条件右边的代码中，参数会被自动转型，所以你不需要去明确地做类型转换。

它还让检测参数否在一个数组范围甚至是集合范围成为可能（我会在这章节的后面讲这个）：

```
val cost = when(x) {
    in 1..10 -> "cheap"
    in 10..100 -> "regular"
    in 100..1000 -> "expensive"
    in specialValues -> "special value!"
    else -> "not rated"
}
```

或者你甚至可以从对参数做需要的几乎疯狂的检查摆脱出来。它可以使用简单的 `if/else` 链替代：

```
valres=when{
    x in 1..10 -> "cheap"
    s.contains("hello") -> "it's a welcome!"
    v is ViewGroup -> "child count: ${v.getChildCount()}"
    else -> ""
}
```

For循环

虽然你在使用了collections的函数操作符之后不会再过多地使用for循环，但是for循环再一些情况下仍然是很有用的。提供一个迭代器它可以作用在任何东西上面：

```
for (item in collection) {
    print(item)
}
```

如果你需要更多使用index的典型的迭代，我们也可以使用 `ranges` （反正它通常是更加智能的解决方案）：

```
for (index in 0..viewGroup.getChildCount() - 1) {
    val view = viewGroup.getChildAt(index)
    view.visibility = View.VISIBLE
}
```

在我们迭代一个array或者list，一系列的index可以用来获取到指定的对象，所以上面的方式不是必要的：

```
for (i in array.indices)
    print(array[i])
```

While和do/while循环

你也可以使用 `while` 循环，尽管它们两个都不是特别常用的。它们通常可以更简单、视觉上更容易理解的方式去解决一个问题，两个例子：

```
while(x > 0){
    x--
}

do{
    val y = retrieveData()
} while (y != null) // y在这里是可见的！
```

Ranges

很难解释 `control flow` ，如果不去讲讲 `ranges` 的话。但是它们的范围要宽得多。 `Range` 表达式使用一个 `..` 操作符，它是被定义实现了一个 `RangTo` 方法。

`Ranges` 帮助我们使用很多富有创造性的方式去简化我们的代码。比如我们可以把它：

```
if(i >= 0 && i <= 10)
    println(i)
```

转化成：

```
if (i in 0..10)
    println(i)
```

`Range` 被定义为可以被比较的任意类型，但是对于数字类型，比较器会通过转换它为简单的类似Java代码来避免额外开销的方式来优化它。数字类型的 `ranges` 也可以被迭代，编译器会转换它们为与Java中使用index的for循环的相同字节码的方式来进行优化：

```
for (i in 0..10)
    println(i)
```

`Ranges` 默认会自增长，所以如果像以下的代码：

```
for (i in 10..0)
    println(i)
```

它就不会做任何事情。但是你可以使用 `downTo` 函数：

```
for(i in 10 downTo 0)
    println(i)
```

我们可以在 `range` 中使用 `step` 来定义一个从1到一个值的不同的空隙：

```
for (i in 1..4 step 2) println(i)

for (i in 4 downTo 1 step 2) println(i)
```

如果你想去创建一个open range（不包含最后一项，译者注：类似数学中的开区间），你可以使用 `until` 函数：

```
for (i in 0 until 4) println(i)
```

这一行会打印从0到3，但是会跳过最后一个值。这也就是说 `0 until 4 == 0..3` 。在一个list中迭代时，使用 `(i in 0 until list.size)` 比 `(i in 0..list.size - 1)` 更加容易理解。

就如之前所提到的，使用 `ranges` 确实有富有创造性的方式。比如，一个简单的方式去从一个 `ViewGroup` 中得到一个Views列表可以这么做：

```
val views = (0..viewGroup.childCount - 1).map { viewGroup.getChildAt(it) }
```

混合使用 `ranges` 和 `函数操作符` 可以避免我们使用明确地循环去迭代一个集合，还有明确地去创建一个我们用来添加views的list。所有的事情都在一行代码中做好了。

如果你想知道更多 `ranges` 的实现方式和更多的范例和游泳的信息，你可以进入Kotlin reference

创建一个详情界面

当我们在主屏幕上点击了一项，我们希望跳转到一个详情界面并且可以看到一些关于那天天气预报的额外信息。我们当前点击了一项之后只是显示了一个toast，但是现在是时候去修改它了。

准备请求

因为我们需要知道哪一个item我们要在详情界面中显示出来，所以逻辑告诉我们需要发送一个天气预报的 `id` 到详情界面。所以 `domain model` 需要一个新的 `id` 属性：

```
data class Forecast(val id: Long, val date: Long, val description
: String,
    val high: Int, val low: Int, val iconUrl: String)
```

`ForecastProvider` 也需要一个新的函数，它返回通过 `id` 请求后的结果。 `DetailActivity` 将需要通过接收到的 `id` 来执行请求获取天气预报数据。因为所有的请求都会迭代所有的数据源并且返回第一个非null的结果，我们可以抽取并定义一个新的函数：

```
private fun <T : Any> requestToSources(f: (ForecastDataSource) -
> T?): T
        = sources.firstResult { f(it) }
```

这个函数使用一个非null类型作为范型。它会接收一个函数，并返回一个可null的对象。其中这个接收的函数接收一个 `ForecastDataSource` ，并返回一个可null范型的对象。我们可以重写上一个请求并如下写一个新的：

```
fun requestByZipCode(zipCode: Long, days: Int): ForecastList = r
equestToSources {
    val res = it.requestForecastByZipCode(zipCode, todayTimeSpan
())
    if (res != null && res.size() >= days) res else null
}

fun requestForecast(id: Long): Forecast = requestToSources {
    it.requestDayForecast(id)
}
```

现在数据源需要去实现一个新的函数：

```
fun requestDayForecast(id: Long): Forecast?
```

`ForcastDb` 将总是会拿到所需的在上一次请求被缓存的值，所以我们可以通过这种方式去获取它：

```
override fun requestDayForecast(id: Long): Forecast? = forecastD
bHelper.use {
    val forecast = select(DayForecastTable.NAME).byId(id).
            parseOpt { DayForecast(HashMap(it)) }
    if (forecast != null) dataMapper.convertDayToDomain(forecast
) else null
}
```

`select` 从查询与之前的非常相似。我创建了另一个名为 `byId` 的工具函数，因为通过 `id` 来请求是很通用的，像这样使用一个函数可以简化处理过程也更具可读性。函数的实现也是相当简单：

```
fun SelectQueryBuilder.byId(id: Long): SelectQueryBuilder
        = whereSimple("_id = ?", id.toString())
```

它只是使用了 `whereSimple` 函数实现使用 `_id` 字段来查询数据。这个函数相当普通，但是如你所见，你可以根据你数据库结构的需要来创建需要的扩展函数，它可以大量地简化你代码的可读性。 `DataMapper` 有一些不值得一提的轻微改变。你可以通过代码库来查看它们。

另一方面， `ForecastServer` 将不会再使用，因为信息总是会被缓存在数据库中。我们可以在一些奇怪的场景下实现一些代码保护，但是我们在这个例子中没有做任何处理，所以如果发生它也会只是抛出一个异常：

```
override fun requestDayForecast(id: Long): Forecast?
        = throw UnsupportedOperationException()
```

> `try` 和 `throw` 是表达式
>
> > 在Kotlin中，几乎一切都是表达式，也就是说一切都会返回一个值。这在函数式编程中是非常重要的，当你使用 `try-catch` 处理边界的问题或者当抛出异常的时候。比如，在上一个例子中，我们可以给结果分配一个exception就算他们不是相同的类型，而不是必须要去创建一个完整的代码块。当我们需要在一个 `when` 分支中抛出一个exception的时候也是非常有用：
> >
> > ```
> > val x = when(y){
> >             in 0..10 -> 1
> >             in 11..20 -> 2
> >             else -> throw Exception("Invalid")
> > }
> > ```
> >
> > `try-catch` 中也是一样，我们可以根据try的结果分配一个值：
> >
> > ```
> > val x = try{ doSomething() }catch{ null }
> > ```

最后一件我们需要做的事就是在新的activity中创建一个command来执行请求：

```
class RequestDayForecastCommand(
        val id: Long,
        val forecastProvider: ForecastProvider = ForecastProvide
r()) :
        Command<Forecast> {
    override fun execute() = forecastProvider.requestForecast(id)

}
```

请求返回一个将用于activity绘制UI的 `Forecast` 结果。

提供一个新的activity

现在我们准备去创建一个 `DetailActivity` 。我们详情activity将会接收一组从主activity传过来的参数： `forecast id` 和 `城市名称` 。第一个参数将会用来从数据库中请求数据，城市名称用于显示在toolbar上。所以我们首先需要定义一组参数的名字：

```
public class DetailActivity : AppCompatActivity() {
    companion object {
        val ID = "DetailActivity:id"
        val CITY_NAME = "DetailActivity:cityName"
    }
    ...
}
```

在 `onCreate` 函数中，第一步是去设置content view。UI是非常简单的，但是对于这个app来说是足够了：

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:gravity="center_vertical"
        tools:ignore="UseCompoundDrawables">
```

```
        <ImageView
            android:id="@+id/icon"
            android:layout_width="64dp"
            android:layout_height="64dp"
            tools:src="@mipmap/ic_launcher"
            tools:ignore="ContentDescription"/>

        <TextView
            android:id="@+id/weatherDescription"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="@dimen/spacing_xlarge"
            android:textAppearance="@style/TextAppearance.AppCom
pat.Display1"
            tools:text="Few clouds"/>
    </LinearLayout>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <TextView
            android:id="@+id/maxTemperature"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:layout_margin="@dimen/spacing_xlarge"
            android:gravity="center_horizontal"
            android:textAppearance="@style/TextAppearance.AppCom
pat.Display3"
            tools:text="30"/>
        <TextView
            android:id="@+id/minTemperature"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:layout_margin="@dimen/spacing_xlarge"
            android:gravity="center_horizontal"
            android:textAppearance="@style/TextAppearance.AppCom
pat.Display3"
            tools:text="10"/>
    </LinearLayout>
```

```
</LinearLayout>
```

然后在 `onCreate` 代码中去设置它。使用城市的名字设置成toolbar的title。 `intent` 和 `title` 通过下面的方法被自动影射到属性：

```
setContentView(R.layout.activity_detail)
title = intent.getStringExtra(CITY_NAME)
```

`onCreate` 实现的另一部分是调用command。这与我们之前做的非常相似：

```
async {
	val result = RequestDayForecastCommand(intent.getLongExtra(ID, -1)).execute()
	uiThread { bindForecast(result) }
}
```

当结果从数据库中获取之后， `bindForecast` 函数在UI线程中被调用。我们在这个activity中又一次使用了Kotlin Android Extensions插件来实现不使用 `findViewById` 来从XML中获取到属性：

```
import kotlinx.android.synthetic.activity_detail.*
...

private fun bindForecast(forecast: Forecast) = with(forecast) {
	Picasso.with(ctx).load(iconUrl).into(icon)
	supportActionBar.subtitle = date.toDateString(DateFormat.FULL)
	weatherDescription.text = description
	bindWeather(high to maxTemperature, low to minTemperature)
}
```

这里有一些有趣的地方。比如，我创建了另一个扩展函数来转换一个 `Long` 对象到一个用于显示的日期字符串。记住我们在adapter中也使用了，所以明确定义它为一个函数是个不错的实践：

```
fun Long.toDateString(dateFormat: Int = DateFormat.MEDIUM): String {
    val df = DateFormat.getDateInstance(dateFormat, Locale.getDefault())
    return df.format(this)
}
```

我会得到一个 `date format` （或者使用默认的DateFormat.MEDIUM）并转换 `Long` 为一个用户可以理解的 `String` 。

另一个有趣的地方是 `bindWeather` 函数。它会接收一个 `vararg` 的由 `Int` 和 `TextView` 组成的 `pairs` ，并且根据温度给 `TextView` 设置不同的 `text` 和 `text color` 。

```
private fun bindWeather(vararg views: Pair<Int, TextView>) = views.forEach {
    it.second.text = "${it.first.toString()}"
    it.second.textColor = color(when (it.first) {
        in -50..0 -> android.R.color.holo_red_dark
        in 0..15 -> android.R.color.holo_orange_dark
        else -> android.R.color.holo_green_dark
    })
}
```

每一个pair，它会设置一个 `text` 来显示温度和一个根据温度匹配的不同的颜色：低温度用红色，中温度用橙色，其它用绿色。温度值是比较随机的，但是这个是使用 `when` 表达式让代码变得简短精炼的很好的代表。

`color` 是我想念的Anko中的一个扩展函数，它可以很简洁的方式从resources中获取一个color，类似于我们在其它地方使用到的 `dimen` 。我们写下这一行的时候，当前 `support library` 依赖 `ContextCompat` 来从不同的Android版本中获取一个color：

```
public fun Context.color(res: Int): Int = ContextCompat.getColor(this, res)
```

`AndroidManifest` 也需要知道新activity的存在：

```
<activity
    android:name=".ui.activities.DetailActivity"
    android:parentActivityName=".ui.activities.MainActivity" >
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value="com.antonioleiva.weatherapp.ui.activities
.MainActivity" />
</activity>
```

启动一个activity：reified函数

最后一步是从 `main activity` 启动一个 `detail activity` 。我们可以如下重写adapter实例：

```
val adapter = ForecastListAdapter(result) {
    val intent = Intent(MainActivity@this, javaClass<DetailActiv
ity>())
    intent.putExtra(DetailActivity.ID, it.id)
    intent.putExtra(DetailActivity.CITY_NAME, result.city)
    startActivity(intent)
}
```

但是这是非常冗长的。一如既往地，Anko提供了简单得多的方式通过 `reified function` 来启动一个activity：

```
val adapter = ForecastListAdapter(result) {
    startActivity<DetailActivity>(DetailActivity.ID to it.id,
            DetailActivity.CITY_NAME to result.city)
}
```

`reified function` 背后到底有什么魔法呢？就像你可能知道的那样，当我们在Java中创建一个范型函数，我们没有办法得到范型类型的Class。一个流行的变通方法是作为参数传入一个Class。在Kotlin中，一个内联（ `inline` ）函数可以被具体化（ `reified` ），这意味着我们可以在函数中得到并使用范型类型的Class。Anko真正使用它的一个简单的例子接下来会讲到（在这个例子中我只使用了 `String` ）：

```
public inline fun <reified T: Activity> Context.startActivity(
        vararg params: Pair<String, String>) {
    val intent = Intent(this, T::class.javaClass)
    params forEach { intent.putExtra(it.first, it.second) }
    startActivity(intent)
}
```

真正的实现要更加复杂一点因为它使用了一个很长的令人讨厌的 `when` 表达式来增加由类型决定的额外信息，但是在概念上来说它没有增加其它更有用的知识。

`Reified` 函数是有一个可以简化代码和提高理解性的语法糖。在这个例子中，它通过获取到了范型类型的 `javaClass` 来创建了一个intent，迭代所有参数并增加到intent，然后使用Intent来启动activity。 `reified` 限制于activity的子类。

剩下的一点细节在代码库中已经说明。我们现在有一个非常简单（但是完整）的主从视图（ `master-detail` ）的App，它使用Kotlin实现，没有使用一行Java代码。

接口

Kotlin中的接口比Java 7中要强大得多。如果你使用Java 8，它们非常相似。在Kotlin中，我们可以像Java中那样使用接口。想象我们有一些动物，它们的其中一些可以飞行。这个是我们针对飞行动物的接口：

```
interface FlyingAnimal {
    fun fly()
}
```

鸟和蝙蝠都可以通过扇动翅膀的方式飞行。所以我们为它们创建两个类：

```
class Bird : FlyingAnimal {
    val wings: Wings = Wings()
    override fun fly() = wings.move()
}

class Bat : FlyingAnimal {
    val wings: Wings = Wings()
    override fun fly() = wings.move()
}
```

当两个类继承自一个接口，非常典型的是它们两者共享相同的实现。但是Java 7中的接口只能定义行为，但是不能去实现它。

Kotlin接口在某一方面它可以实现函数。它们与类唯一的不同之处是它们是无状态（stateless）的，所以属性需要子类去重写。类需要去负责保存接口属性的状态。

我们可以让接口实现 `fly` 函数：

```
interface FlyingAnimal {
    val wings: Wings
    fun fly() = wings.move()
}
```

就像提到的那样，类需要去重写属性：

```
class Bird : FlyingAnimal {
    override val wings: Wings = Wings()
}

class Bat : FlyingAnimal {
    override val wings: Wings = Wings()
}
```

现在鸟和蝙蝠都可以飞行了：

```
val bird = Bird()
val bat = Bat()

bird.fly()
bat.fly()
```

委托

委托模式是一个很有用的模式，它可以用来从类中抽取出主要负责的部分。委托模式是Kotlin原生支持的，所以它避免了我们需要去调用委托对象。委托者只需要指定实现的接口的实例。

在我们前面的例子中，我们可以通过构造函数指定动物怎么飞行，而不是实现它。比如，一个使用翅膀飞行的动物可以用这种方式指定：

```
interface CanFly {
    fun fly()
}

class Bird(f: CanFly) : CanFly by f
```

我们可以使用接口来指示鸟可以飞行，但是鸟的飞行方式被定义在一个委托中，这个委托定义在构造函数中，所以我们可以针对不同的鸟使用不同的飞行方式。动物使用翅膀飞行的方式被定义在另一个类中：

```
class AnimalWithWings : CanFly {
    val wings: Wings = Wings()
    override fun fly() = wings.move()
}
```

动物扇动翅膀来飞行。所以我们可以创建一个鸟，它使用翅膀飞行：

```
val birdWithWings = Bird(AnimalWithWings())
birdWithWings.fly()
```

但是现在翅膀可以被别的不是鸟类的动物使用。如果我们假设蝙蝠使用翅膀，我们可以直接指定委托来实例化对象：

```
class Bat : CanFly by AnimalWithWings()
...
val bat = Bat()
bat.fly()
```

在我们的App中实现一个例子

接口可以被用来从类中提取出相似行为的通用代码。比如，我们可以创建一个接口用于处理app的toolbar。`MainActivity` 和 `DetailActivity` 在处理 `toolbar` 时会共享这些相似的代码。

但是受限，我们需要做出一些改变，使用被定义在布局中 `toolbar` ，而不是标准的 `ActionBar` 。第一件事是继承 `NoActionBar` 主题。这样 `toolbar` 不会自动被包含进来：

```
<style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar
">
    <item name="colorPrimary">#ff212121</item>
    <item name="colorPrimaryDark">@android:color/black</item>
</style>
```

我们使用 `light` 主题。然后我们创建一个 `toolbar` 的布局，我们稍后会在其它的布局中使用到它：

```
<android.support.v7.widget.Toolbar
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    app:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>
```

`toolbar` 指定了它自己的背景，一个针对自己的 `dark` 主题和一个针对生成的弹出框的 `light` 主题（ `overflow menu` 实例）。我们现在已经有了相同的主题： `light` 主题和 `dark` 主题的 `Action Bar` 。

下一步我们将修改 `MainActivity` 的布局，增加一个toolbar：

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/forecastList"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:clipToPadding="false"
        android:paddingTop="?attr/actionBarSize"/>

    <include layout="@layout/toolbar"/>
</FrameLayout>
```

现在toolbar被增加到布局中，我们可以开始使用它。我们创建了一个接口，它可以让我们：

- 改变title
- 指定是否显示上一步的导航动作
- 滚动时的toolbar动画
- 给所有的activity设置相同的菜单，甚至行为

然后让我们定义 `ToolbarManager` ：

```
interface ToolbarManager {
    val toolbar: Toolbar
    ...
}
```

它将需要一个toolbar属性。接口是无状态的，所以属性可以被定义，但是不能赋值。子类会实现这个接口并重写这个属性。

另一方面，我们可以不使用重写来实现无状态的属性。也就是说属性不需要维护一个 `backup field` 。一个处理toolbar title属性的例子：

```
var toolbarTitle: String
    get() = toolbar.title.toString()
    set(value) {
        toolbar.title = value
    }
```

因为属性仅仅使用了toolbar，它不需要保存任何新的状态。

我们现在创建了一个新的函数用来初始化toolbar，inflate一个menu并且设置一个listener：

```
fun initToolbar(){
    toolbar.inflateMenu(R.menu.menu_main)
    toolbar.setOnMenuItemClickListener {
        when (it.itemId) {
            R.id.action_settings -> App.instance.toast("Settings"
)
            else -> App.instance.toast("Unknown option")
        }
        true
    }
}
```

我们可以增加一个函数用来开启toolbar上面导航icon，设置一个箭头的icon并设置一个当icon被按压时触发的事件：

```
fun enableHomeAsUp(up: () -> Unit) {
       toolbar.navigationIcon = createUpDrawable()
       toolbar.setNavigationOnClickListener { up() }
}

private fun createUpDrawable() = with (DrawerArrowDrawable(toolb
ar.ctx)){
    progress = 1f
    this
}
```

这个函数接收一个listener，使用DrawerArrowDrawable来创建一个最后状态（当箭头已经显示时）的drawable，然后把listener设置给toolbar。

最后，接口将会提供一个函数，它允许toolbar可以attached到一个scroll上面，并且根据scroll的方向来执行动画。当往下滚动时toolbar会消失 ，往上滚动toolbar会再次显示：

```
fun attachToScroll(recyclerView: RecyclerView) {
    recyclerView.addOnScrollListener(object : RecyclerView.OnScr
ollListener() {
        override fun onScrolled(recyclerView: RecyclerView?, dx:
 Int, dy: Int) {
            if (dy > 0) toolbar.slideExit() else toolbar.slideEn
ter()
        }
    })
}
```

我们会创建两个用于view从屏幕中显示或者消失动画的扩展函数。我们会检查是否动画之前没有执行过。这种方式可以避免每次不同的滚动view都会执行动画：

```
fun View.slideExit() {
    if (translationY == 0f) animate().translationY(-height.toFla
t())
}

fun View.slideEnter() {
    if (translationY < 0f) animate().translationY(0f)
}
```

在 `toobar manager` 实现之后，是时候在 `MainActivity` 中使用它了。我们首先指定toolbar属性。我们可以使用 `lazy` 委托实现，这样会在我们第一次使用它的时候才会inflate：

```
override val toolbar by lazy { find<Toolbar>(R.id.toolbar) }
```

`MainActivity` 将会仅仅初始化toolbar并attach到 `RecyclerView` 的滚动并修改toolbar的title：

```
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main) initToolbar()
    forecastList.layoutManager = LinearLayoutManager(this)
    attachToScroll(forecastList)
    async {
        val result = RequestForecastCommand(94043).execute()
        uiThread {
            val adapter = ForecastListAdapter(result) {
            startActivity<DetailActivity>(DetailActivity.ID to i
t.id,
                        DetailActivity.CITY_NAME to result.city)
            }
            forecastList.adapter = adapter
            toolbarTitle = "${result.city} (${result.country})"
        }
    }
}
```

`DetailActivity` 也需要一些布局上的修改：

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <include layout="@layout/toolbar"/>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:gravity="center_vertical"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        tools:ignore="UseCompoundDrawables">
       ....
    </LinearLayout>

</LinearLayout>
```

使用相同的方式去指定toolbar属性。 `DetailActivity` 也会初始化toolbar，设置title并且开启导航返回icon：

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_detail)

    initToolbar()
    toolbarTitle = intent.getStringExtra(CITY_NAME)
    enableHomeAsUp { onBackPressed() }
    ...
}
```

接口可以帮助我们从类中提取出公共的代码来共享相似的行为。可以作为让我们代码精炼合理简洁可复用的替代方案。思考哪方面接口可以帮助你写出更好的代码。

泛型

泛型编程包括，在不指定代码中使用到的确切类型的情况下来编写算法。用这种方式，我们可以创建函数或者类型，唯一的区别只是它们使用的类型不同，提高代码的可重用性。这种代码单元就是我们所知道的泛型，它们存在于很多的语言之中，包括Java和Kotlin。

在Kotlin中，泛型甚至更加重要，因为经常使用扩展函数将会成倍增加我们泛型使用频率。尽管我们已经在本书中盲目地使用了泛型，但是泛型在任何语言中通常都是比较困难的一部分，所以我尝试使用尽可能简单的方式来讲解它，这样主要的思想也会足够地清晰。

基础

举个例子，我们可以创建一个指定泛型类：

```
class TypedClass<T>(parameter: T) {
    val value: T = parameter
}
```

这个类现在可以使用任何的类型初始化，并且参数也会使用定义的类型，我们可以这么做：

```
val t1 = TypedClass<String>("Hello World!")
val t2 = TypedClass<Int>(25)
```

但是Kotlin很简单并且缩减了模版代码，所以如果编译器能够推断参数的类型，我们甚至也就不需要去指定它：

```
val t1 = TypedClass("Hello World!")
val t2 = TypedClass(25)
val t3 = TypedClass<String?>(null)
```

如第三个对象接收一个null引用，那仍然还是需要指定它的类型，因为它不能去推断出来。

我们可以像Java中那样在定义中指定的方式来增加类型限制。比如，如果我们想限制上一个类中为非null类型，我们只需要这么做：

```
class TypedClass<T : Any>(parameter: T) {
    val value: T = parameter
}
```

如果你再去编译前面的代码，你将看到 `t3` 现在会抛出一个错误。可null类型不再被允许了。但是限制明显可以更加严厉。如果我们只希望 `Context` 的子类该怎么做？很简单：

```
class TypedClass<T : Context>(parameter: T) {
    val value: T = parameter
}
```

现在所有继承 `Context` 的类都可以在我们这个类中使用。其它的类型是不被允许的。

当然，可以使用函数中。我们可以相当简单地构建泛型函数：

```
fun <T> typedFunction(item: T): List<T> {
    ...
}
```

变体

这是真的是最难理解的部分之一。在Java中，当我们使用泛型的时候会出现问题。逻辑告诉我们 `List<String>` 应该可以转型为 `List<Object>` ，因为它有更弱的限制。但是我们来看下这个例子：

```
List<String> strList = new ArrayList<>();
List<Object> objList = strList;
objList.add(5);
String str = objList.get(0);
```

如果Java编译器允许我们这么做，我们可以增加一个 `Integer` 到 `Object` List，但是它明显会在某一时刻奔溃。这就是为什么语言中增加了通配符。通配符可以在限制这个问题中可以增加灵活性。

如果我们增加了 `? extends Object` ，我们使用了协变（ `covariance` ），它表示我们可以处理任何使用了类型，比Object更严格的对象，但是我们只有使用 `get` 操作时是安全的。如果我们想去拷贝一个 `Strings` 集合到 `Objects` 集合中，我们应该是允许的，对吧？然后，如果我们这样：

```
List<String> strList = ...;
List<Object> objList = ...;
objList.addAll(strList);
```

这样是可以的，因为定义在 `Collection` 接口中的 `addAll()` 是这样的：

```
List<String>
interface Collection<E> ... {
    void addAll(Collection<? extends E> items);
}
```

否则，没有通配符，我们不会允许在这个方法中使用 `String` List。相反地，当然会失败。我们不能使用 `addAll()` 来增加一个 `Objects` List到 `Strings` List中。因为我们只是用那个方法从 `collection` 中获取元素，这是一个完美的协变（ `covariance` ）的例子。

另一方面，我们可以在对立面上发现逆变（ `contravariance` ）。按照集合的例子，如果我们想把传过来的参数增加到集合中去，我们可以增加更加限制的类型到泛型集合中。比如，我们可以增加 `Strings` 到 `Object` List：

```
void copyStrings(Collection<? super String> to, Collection<Strin
g> from) {
    to.addAll(from);
}
```

增加 `Strings` 到另一个集合中唯一的限制就是那个集合接收 `Strings` 或者父类。

但是通配符都有它的限制。通配符定义了使用场景变体（ `use-site variance` ），这意味着当我们使用它的时候需要声明它。这表示每次我们声明一个泛型变量时都会增加模版代码。

让我们看一个例子。使用我们之前相似的类：

```
class TypedClass<T> {
    public T doSomething(){
        ...
    }
}
```

这些代码不会被编译：

```
TypedClass<String> t1 = new TypedClass<>();
TypedClass<Object> t2 = t1;
```

尽管它的确没有意义，因为我们仍然保持了类中的所有的方法并且没有任何损坏。我们需要指定的类型可以有一个更加灵活的定义。

```
TypedClass<String> t1 = new TypedClass<>();
TypedClass<? extends String> t2 = t1;
```

这会让代码更加难以理解，而且增加了一些额外的模版代码。

另一方面，Kotlin通过内部声明变体（`declaration-site variance`）可以使用更加容易的方式来处理。这表示当我们定义一个类或接口的时候我们可以处理弱限制的场景，我们可以在其它地方直接使用它。

所以让我们看看它在Kotlin中是怎么工作的。相比冗长的通配符，Kotlin仅仅使用`out`来针对协变（`covariance`）和使用`in`来针对逆变（`contravariance`）。在这个例子中，当我们类产生的对象可以被保存到弱限制的变量中，我们使用协变。我们可以直接在类中定义声明{

```
class TypedClass<out T>() {
    fun doSomething(): T {
        ...
    }
}
```

这就是所有我们需要的。现在，在Java中不能编译的代码在Kotlin中可以完美运行：

```
val t1 = TypedClass<String>()
val t2: TypedClass<Any> = t1
```

如果你已经使用了这些概念，我确信你可以很简单地在Kotlin使用`in`和`out`。否则，你也只是需要一些联系和概念上的理解。

泛型例子

理论之后，我们转移到一些实际功能上面，这会让我们更加简单地掌握它。为了不重复发明轮子，我使用三个Kotlin标准库中的三个函数。这些函数让我们仅使用泛型的实现就可以做一些很棒的事情。它可以鼓舞你创建自己的函数。

let

`let` 实在是一个简单的函数，它可以被任何对象调用。它接收一个函数（接收一个对象，返回函数结果）作为参数，作为参数的函数返回的结果作为整个函数的返回值。它在处理可null对象的时候是非常有用的，下面是它的定义：

```
inline fun <T, R> T.let(f: (T) -> R): R = f(this)
```

它使用了两个泛型类型： `T` 和 `R` 。第一个是被调用者定义的，它的类型被函数接收到。第二个是函数的返回值类型。

我们怎么去使用它呢？你可能还记得当我们从数据源中获取数据时，结果可能是null。如果不是null，则把结果映射到 `domain model` 并返回结果，否则直接返回null：

```
if (forecast != null) dataMapper.convertDayToDomain(forecast) el
se null
```

这代码是非常丑陋的，我们不需要使用这种方式去处理可null对象。实际上如果我们使用 `let` ，都不需要 `if` ：

```
forecast?.let { dataMapper.convertDayToDomain(it) }
```

多亏 `?.` 操作符， `let` 函数只会在 `forecast` 不是null的时候才会执行。否则它会返回null。也就是我们想达到的效果。

with

本书中我们大量讲了这个函数。 `with` 接收一个对象和一个函数，这个函数会作为这个对象的扩展函数执行。这表示我们根据推断可以在函数内使用 `this` 。

```
inline fun <T, R> with(receiver: T, f: T.() -> R): R = receiver.f
()
```

泛型在这里也是以相同的方式运行： `T` 代表接收类型， `R` 代表结果。如你所见，函数通过 `f: T.() -> R` 声明被定义成了扩展函数。这就是为什么我们可以调用 `receiver.f()` 。

通过这个app，我们有几个例子：

```
fun convertFromDomain(forecast: ForecastList) = with(forecast) {
    val daily = dailyForecast map { convertDayFromDomain(id, it)
 }
    CityForecast(id, city, country, daily)
}
```

apply

它看起来于 `with` 很相似，但是是有点不同之处。 `apply` 可以避免创建builder的方式来使用，因为对象调用的函数可以根据自己的需要来初始化自己，然后 `apply` 函数会返回它同一个对象：

```
inline fun <T> T.apply(f: T.() -> Unit): T { f(); return this }
```

这里我们只需要一个泛型类型，因为调用这个函数的对象也就是这个函数返回的对象。一个不错的例子：

```
val textView = TextView(context).apply {
    text = "Hello"
    hint = "Hint"
    textColor = android.R.color.white
}
```

它创建了一个 `TextView` ，修改了一些属性，然后赋值给一个变量。一切都很简单，具有可读性和坚固的语法。让我们用在当前的代码中。

在 `ToolbarManager` 中，我们使用这种方式来创建导航drawable：

```
private fun createUpDrawable() = with(DrawerArrowDrawable(toolbar.ctx)) {
    progress = 1f
    this
}
```

使用 `with` 和返回 `this` 是非常清晰的，但是使用 `apply` 可以更加简单：

```
private fun createUpDrawable() = DrawerArrowDrawable(toolbar.ctx).apply {
    progress = 1f
}
```

你可以在 `Kotlin for Android Developer` 代码库中查看这些小的优化。

设置界面

直到现在，我们都是使用的默认的城市来实现这个app，但是现在是时候增加一个选择城市的功能了。我们的App需要一个设置栏来让用户修改城市。

我们使用 `zip code` （邮编）来区分城市。一个真正的App可能需要更多的信息，因为只有邮编在整个世界中可能无法作为辨认依据。但是我们至少会显示在设置中使用 `zip code` 定义的世界上的城市。这会是一个用来解释怎么使用有趣的方式处理preferences的例子。

创建一个设置activity

当toolbar上溢出菜单（ `overflow menu` ）的 `settings` 选项被点击时，需要打开一个新的Activity。所以首先要做的事情时需要一个新的 `SettingActivity` ：

```
class SettingsActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_settings)
        setSupportActionBar(toolbar)
        supportActionBar.setDisplayHomeAsUpEnabled(true)
    }

    override fun onOptionsItemSelected(item: MenuItem) = when (i
tem.itemId) {
        android.R.id.home -> { onBackPressed(); true }
        else -> false
    }
}
```

当用户离开这个界面的时我们需要保存用户 `preference` （偏好），所以我们需要像处理 `Back` 一样处理 `Up` 动作，重定向动作到 `onBackPressed` 。现在，让我们要创建一个XML布局。对于这个 `preference` 来说一个简单 `EditText` 就足够了：

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <include layout="@layout/toolbar"/>
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="?attr/actionBarSize"
        android:padding="@dimen/spacing_xlarge">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/city_zipcode"/>

        <EditText
            android:id="@+id/cityCode"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="@string/city_zipcode"
            android:inputType="number"/>

    </LinearLayout>
</FrameLayout>
```

然后只需要在 `AndroidManifest.xml` 中声明这个activity：

```
<activity
android:name=".ui.activities.SettingsActivity"
android:label="@string/settings"/>
```

访问Shared Preferences

你可能知道什么是Android Shared Preferences。可以通过Android框架简单存储的一系列key和value对。这些 `preferences` 与SDK的一部分融为一体，使得任务变得更加容易。而且从Android 6.0（Marshmallow）， `shared preferences` 可以自动被云存储，所以当一个用户在一个新的设备上面恢复App的时候，它们的 `preferences` 也会被恢复。

多亏使用了属性委托，我们可以使用非常简单的方式来处理 `preferences` 。我们可以创建一个委托，当 `get` 被调用时去查询，当 `set` 被调用时去执行保存操作。

因为我们想去保存 `zip code` ，它是一个long型，所以让我们创建一个Long属性的委托吧。在 `DelegatesExtensions.kt` 中，实现一个新的 `LongPreference` 类：

```
class LongPreference(val context: Context, val name: String, val
 default: Long)
    :  ReadWriteProperty<Any?, Long> {

    val prefs by lazy {
        context.getSharedPreferences("default", Context.MODE_PRI
VATE)
    }

    override fun getValue(thisRef: Any?, property: KProperty<*>)
: Long {
        return prefs.getLong(name, default)
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>,
 value: Long) {
        prefs.edit().putLong(name, value).apply()
    }
}
```

首先，我们使用 `lazy` 委托的方式创建一个preferences。这样的话，如果我们没有使用这个属性，这个委托就不会请求这个 `SharedPreferences` 对象。

当 `get` 被调用，它的实现是使用preferences实例去获取一个委托声明中指定名字的long属性，如果没有找到这个属性，则默认使用default。当一个值被 `set` ，拿到 `preferences editor` 并使用属性名保存。

我们可以在 `DelegatesExt` 中定义一个新的委托，这样我们访问时就简单很多：

```
object DelegatesExt {
    ....
    fun longPreference(context: Context, name: String, default:
Long) =
        LongPreference(context, name, default)
}
```

在 `SettingActivity` ，现在可以定义一个属性去处理 `zip code` 偏好。我创建了两个常量用来作为名字和属性的默认值。这种方式可以在App其他地方使用：

```
companion object {
    val ZIP_CODE = "zipCode"
    val DEFAULT_ZIP = 94043L
}

var zipCode: Long by DelegatesExt.longPreference(this, ZIP_CODE,
 DEFAULT_ZIP)
```

现在preference工作起来就非常简单了，我们可以从属性中得到并赋值给 `EditText` ：

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    cityCode.setText(zipCode.toString())
}
```

我们不能使用自动生成的属性 `text` ，因为 `EditText` 在 `getText` 中返回的是 `Editable` ，所以该属性默认为该值。如果我尝试去分配一个 `String` ，编译器会报错，使用 `setText()` 就足够了。

现在具备了所有要实现 `onBackPressed` 的东西。这里，一个属性的新值会被储存：

```
override fun onBackPressed() {
    super.onBackPressed()
    zipCode = cityCode.text.toString().toLong()
}
```

`MainActivity` 需要一些小的改变。首先，它也需要一个 `zip code` 属性。

```
val zipCode: Long by DelegatesExt.longPreference(this, SettingsA
ctivity.ZIP_CODE,
            SettingsActivity.DEFAULT_ZIP)
```

然后，我把 `forecast load` 的代码移动到了 `onResume` ，这样每次activity resumed，它都会刷新数据，以防 `code zip` 被修改。当然这里有更加复杂一点的方式去做，比如通过在请求forecast之前检查是否 `zip code` 真的改变了。但是我像保持这个例子的简单性，而且因为请求的数据已经保存在本地数据库中了，所以这个解决方案也不算太坏：

```
override fun onResume() {
    super.onResume()
    loadForecast()
}

private fun loadForecast() = async {
    val result = RequestForecastCommand(zipCode).execute()
    uiThread {
        val adapter = ForecastListAdapter(result) {
            startActivity<DetailActivity>(DetailActivity.ID to i
t.id,
                    DetailActivity.CITY_NAME to result.city)
        }
        forecastList.adapter = adapter
        toolbarTitle = "${result.city} (${result.country})"
    }
}
```

`RequestForecastCommand` 现在使用 `zipCode` 而不是之前的是一个固定值。

这里还有意见我们必须要做的事情：当溢出菜单的 `settings` 点击时启动这个 `setting activity` 。在 `ToolbarManager` 中的 `initToolbar` 函数需要有一些小的修改：

```
when (it.itemId) {
    R.id.action_settings -> toolbar.ctx.startActivity<SettingsAc
tivity>()
    else -> App.instance.toast("Unknown option")
}
```

泛型preference委托

现在我们已经是泛型专家了，为什么不扩展 `LongPreference` 为支持所有 `Shared Preferences` 支持的类型呢？我们来创建一个 `Preference` 委托：

```
class Preference<T>(val context: Context, val name: String, val
default: T)
    : ReadWriteProperty<Any?, T> {

    val prefs by lazy {
        context.getSharedPreferences("default", Context.MODE_PRI
VATE)
    }

    override fun getValue(thisRef: Any?, property: KProperty<*>)
: T {
        return findPreference(name, default)
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>,
 value: T) {
        putPreference(name, value)
    }
    ...
}
```

这个preference与我们之前使用的非常相似。我们仅仅替换了 `Long` 为泛型类型 `T` ，然后调用了两个函数来做具体重要的工作。这些函数非常简单，尽管有些重复。它们会检查类型然后使用指定的方式来操作。比如， `findPrefernce` 函数如下：

```
private fun <T> findPreference(name: String, default: T): T = wi
th(prefs) {
        val res: Any = when (default) {
        is Long -> getLong(name, default)
        is String -> getString(name, default)
        is Int -> getInt(name, default)
        is Boolean -> getBoolean(name, default)
        is Float -> getFloat(name, default)
        else -> throw IllegalArgumentException(
            "This type can be saved into Preferences")
    }

    res as T
}
```

`putPreference` 函数也是一样，但是在 `when` 最后通过 `apply` ，使用 `preferences editor` 保存结果：

```
private fun <U> putPreference(name: String, value: U) = with(pre
fs.edit()) {
    when (value) {
        is Long -> putLong(name, value)
        is String -> putString(name, value)
        is Int -> putInt(name, value)
        is Boolean -> putBoolean(name, value)
        is Float -> putFloat(name, value)
        else -> throw IllegalArgumentException("This type can be
 saved into Pref\
erences")
    }.apply()
}
```

现在修改 `DelegateExt` ：

```
object DelegatesExt {
    ...
    fun preference<T : Any>(context: Context, name: String, defa
ult: T)
        = Preference(context, name, default)
}
```

这章之后，用户可以访问设置界面并修改 `zip code` 。然后当我们返回主界面，forecast会自动重新刷新并显示新的信息。查看代码库中其余xi wei

测试你的App

我们即将到达这次旅程的结尾。通过本书你已经学习了大部分Kotlin的知识，但是你可能会怀疑你是否可以测试你只用Kotlin编写的Android App呢？回答是：当然！

在Android中我们有两种不同的测试：`unit test`和`instrumentation test`。很明显本书不会来教你怎么去测试的，有很多专门为此写的书。我在这一章的目标是怎么去搭建你测试环境，展示给你看Kotlin在测试方面也能很好的工作。

Unit testing

我不会对 `unit testing` （单元测试）是什么的话题展开讨论。存在很多定义，但是都有一些细微的不同。一个普通的观点可能是 `unit testing` 验证一个单位（ `unit` ）的源代码的测试。一个单位（ `unit` ）包含什么就留给读者了。在我们的例子中，我仅仅去定义了一个 `unit test` 作为一个不需要设备运行的测试。IDE将会运行这些测试然后显示最后的结果分辩哪些测试成功哪些测试失败了。

`Unit testing` 通常使用 `JUnit` 库。所以让我们增加这个依赖到 `build.gradle` 。因为这个依赖只会在跑测试的时候才会用到，所以我们可以使用 `testCompile` 而不是 `compile` 。用这种方式，这个库会在正式编译时忽略掉，可以减少APK的大小：

```
dependencies {
    ...
    testCompile 'junit:junit:4.12'
}
```

现在同步gradle来获取该库并加入到你的项目中。为了开启 `unit testing` ，打开 `Build Variants` tab（你可能可以在IDE的左边找到它），点击 `Test Artifact` 下拉，你应该选择 `Unit Tests` 。

另一件你需要做的事情是创建一个新的文件夹。在src下面，你可能已经有 `androidTest` 和 `main` 了。创建另一个名为 `test` 的文件夹，再在它下面创建一个 `java` 文件夹。所以现在你应该有一个名为 `src/test/java` 绿色的文件夹。这是IDE发现我们在使用 `Unit Test` 模式好的迹象，这个文件夹中将会包括一些测试文件。

我们来写一个非常简单的测试来看看一切是不是正常运行了。使用合适的包名（我的是 `com.antonioleiva.weatherapp` ，但是你需要使用你app中的主包名）创建一个新的名为 `SimpleTest` 的Kotlin类。当你创建完，编写如下简单的测试：

```
import org.junit.Test
import kotlin.test.assertTrue
class SimpleTest {
    @Test fun unitTestingWorks() {
        assertTrue(true)
    }
}
```

使用 `@Test` 注解来辨别该函数为是一个测试。确认是 `org.unit.Test` 。然后增加一个简单的断言。它只是判断了true是否是true，它显然会成功。这个测试只是用开确认一切配置正确。

执行测试，只需要在你在 `test` 下创建的新的 `java` 文件夹上右击，然后选择 `Run All Tests` 。当编译完成后，它会运行测试并会看见结果简介的显示。你应该可以看见我们的测试通过了。

现在是时候创建一个真正的测试了。所有使用Android框架来处理的测试可能都需要一个 `instrumentation test` 或者使用更复杂的像Robolectric库。所以在这些例子中我会不使用框架的任何东西。举个例子，我将测试从 `Long` 转 `String` 的扩展函数。

创建一个新的名为 `ExtensionTests` 的文件，然后增加如下测试：

```
class ExtensionsTest {

    @Test fun testLongToDateString() {
        assertEquals("Oct 19, 2015", 1445275635000L.toDateString
())
    }

    @Test fun testDateStringFullFormat() {
        assertEquals("Monday, October 19, 2015",
            1445275635000L.toDateString(DateFormat.FULL))
    }
}
```

这些测试检测 `Long` 实例是否可以转换成一个 `String` 。第一个测试默认行为（使用DateFormat.MEDIUM)），而第二个指定一个不同的格式。运行这些测试然后你会看到它们都通过了。我建议你修改它们然后看看它们失败是怎么样的。

如果你在Java中使用过测试，你将会发现这并没有什么太多的不同。我会演示一个简单的例子，我们可以对 `ForecastProvider` 进行一些测试。我们可以使用 `Mockito` 库来模拟其它的类然后独立测试provider：

```
dependencies {
    ...
    testCompile "junit:junit:4.12"
    testCompile "org.mockito:mockito-core:1.10.19"
}
```

现在创建了一个 `ForecastProviderTest` 。我们要去测试 `ForecastProvider` ，使用 `DataSource` 来返回结果，看它结果是否为null。所以首先我们需要模拟一个 `ForecastDataSource` ：

```
val ds = mock(ForecastDataSource::class.java)
`when`(ds.requestDayForecast(0)).then {
    Forecast(0, 0, "desc", 20, 0, "url")
}
```

如你所见，我们需要在 `when` 上加反引号。因为 `when` 在Kotlin中是一个保留关键字，所以如果我们在一些Java代码中使用到它我们需要避免它。现在我们用这个数据源创建了一个provider，然后检测调用那个方法之后的结果是否为null：

```
val provider = ForecastProvider(listOf(ds))
assertNotNull(provider.requestForecast(0))
```

这是完整的测试函数：

```
@Test fun testDataSourceReturnsValue() {
    val ds = mock(ForecastDataSource::class.java)
    `when`(ds.requestDayForecast(0)).then {
           Forecast(0, 0, "desc", 20, 0, "url")
    }

    val provider = ForecastProvider(listOf(ds))
    assertNotNull(provider.requestForecast(0))
}
```

如果你运行它，你将会看见它会出错。多亏这个测试，我们在自己的代码中发现了某些错误。测试失败是因为 `ForecastProvider` 在使用之前正在它的 `companion object` 中初始化。我们可以通过构造函数的方式在 `ForecastProvider` 中增加一些数据源，这个静态的List就永远不会被使用，所以它应该是使用 `lazy` 加载：

```
companion object {
    val DAY_IN_MILLIS = 1000 * 60 * 60 * 24
    val SOURCES by lazy { listOf(ForecastDb(), ForecastServer())
 }
}
```

如果你现在再次去运行，你会发现现在会通过所有的测试。 我们也可以测试一些比如当数据源返回null的时候，它会便利下一个数据源来得到结果：

```
@Test fun emptyDatabaseReturnsServerValue() {
    val db = mock(ForecastDataSource::class.java)
    val server = mock(ForecastDataSource::class.java)
    `when`(server.requestForecastByZipCode(
            any(Long::class.java), any(Long::class.java)))
            .then {
                ForecastList(0, "city", "country", listOf())
    val provider = ForecastProvider(listOf(db, server))
    assertNotNull(provider.requestByZipCode(0, 0))
}
```

如你所见，通过使用参数的默认值这种简单的依赖倒置足够让我们实现一些简单的 `unit tests` 。对于这个provider还有很多我们可以测试的东西，但是这个例子足够让我们学会使用 `unit testing` 工具了。

Instrumentation tests

`Instrumentation tests` 有一点不同。它们通常被使用在UI交互上，我们需要一个应用程序实例跑的同时执行测试。达到这个，我们就需要在设备上部署并运行。

这类的测试必须要放在 `androidTest` 文件夹中，我们必须要修改 `Build Variants` 区域的 `Test Artifact` 为 `Android Instrumentation Tests` 。实现instrumentation的官方库是Espresso，它通过 `Actions` 、 `filter` 以及检测结果的 `ViewMatchers` 和 `Matchers` 可以帮助我们更简单地使用。

配置比之前更加难一点。我们需要下载额外的库和 `Gradle` 的配置。好事是Kotlin的测试不需要添加额外的东西，所以如果你已经知道怎么去配置 `Espresso` ，它将对你来说是很简单的。

首先，在 `defaultConfig` 中指定 `test runner` ：

```
defaultConfig {
    ...
    testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
}
```

当你处理完该runner，然后增加其它的依赖，这次是用 `androidTestCompile` 。这种方式，这些库只会再编译运行 `instrumentation tests` 的时候才被增加：

```
dependencies {
    ...
    androidTestCompile "com.android.support:support-annotations:$support_version"
    androidTestCompile "com.android.support.test:runner:0.4.1"
    androidTestCompile "com.android.support.test:rules:0.4.1"
    androidTestCompile "com.android.support.test.espresso:espresso-core:2.2.1"
    androidTestCompile ("com.android.support.test.espresso:espresso-contrib:2.2.1"){
        exclude group: 'com.android.support', module: 'appcompat'

        exclude group: 'com.android.support', module: 'support-v4'
        exclude module: 'recyclerview-v7'
}
```

我不想花大量的时间去讲这些，但是这里有为什么需要这些库的简短原因：

- `support-annotations` ：其它库中需要使用到。
- `runner` ：这是 `test runner` ，就是我们再 `defaultConfig` 中指定的那个。
- `rules` ：包括一些测试inflate启动activity的规则。我们将会在我们的例子中使用这些规则。
- `espresso-core` ： `Espresso` 的基本实现，它让 `instrument tests` 更加容易。
- `espresso-contrib` ：它增加了其它额外的功能，比如支持 `RecyclerView` 测试。我们不得不排除掉一些它的依赖，因为我们已经在这个项目中使用到了，否则测试会出错。

我们现在来创建一个简单的例子。测试将会点击forecast列表的第一行，然后它会判断是否能找到一个id为 `R.id.weatherDescription` 的view。这个view是在 `DetailActivity` 中的，这表示我们在测试在RecyclerView里面点击后是否可以成功地导航到详情页面。

```
class SimpleInstrumentationTest {

    @get:Rule
    val activityRule = ActivityTestRule(MainActivity::class.java)


    ...
}
```

首先我们需要指定它运行时使用 `AndroidJUnit4` 。然后，创建一个activity规则，它会实例化一个测试需要的activity。在Java中，你可以使用 `@Rule` 。但是如你所知，字段和属性是不一样的，所以如果你像那样去使用的话，执行会失败因为访问属性中的字段是不是public的。你需要加注解的是在getter上面。Kotlin允许指定 `get` 或者 `set` 在Rule的名字前面。在这个例子中，只要些 `@get:Rule` 。

之后，我们已经准备好创建第一个测试了：

```
@Test fun itemClick_navigatesToDetail() {
    onView(withId(R.id.forecastList)).perform(
            RecyclerViewActions
                .actionOnItemAtPosition<RecyclerView.ViewHolder>(
0, click()))
    onView(withId(R.id.weatherDescription))
            .check(matches(isAssignableFrom(TextView::class.java)
))
}
```

函数加上了 `@Test` 注解，这根我们使用 `unit test` 的方式一样。我们可以开始在测试体中使用 `Espresso` 。它首先在 `RecyclerView` 的第一个position中执行了一个点击。然后它检测是否可以找到一个指定id的view且这个view是一个 `TextView` 。

要运行这个测试，点击顶部的 `Run configurations` 下拉选择 `Edit Configurations...` 按下 `+` 图标，选择 `Android Tests` ，然后选择 `app` 模块。现在，在 `target device` 中选择你喜欢的 `target` 。点击 `OK` 然后运行。你

应该可以看到App是怎样在你的设备中开始的，它会测试第一个position，打开详情页面然后再次关闭app。

现在我们要做一个更加复杂一点的事情。测试会从toolbar众打开一个溢出菜单，点击 `settings` 栏，改变城市的 `code` ，然后检测toolbar的标题是否改变成了对应的标题。

```
@Test fun modifyZipCode_changesToolbarTitle() {
    openActionBarOverflowOrOptionsMenu(activityRule.activity)
    onView(withText(R.string.settings)).perform(click())
    onView(withId(R.id.cityCode)).perform(replaceText("28830"))
    pressBack()
    onView(isAssignableFrom(Toolbar::class.java))
            .check(matches(
                withToolbarTitle(`is`("San Fernando de Henares (
ES)"))))
}
```

这个测试实际做的事情：

- 它首先使用 `openActionBarOverflowOrOptionsMenu` 打开溢出菜单。
- 然后它根据 `Settings` 文本查找一个view，然后点击这个它。
- 之后，设置界面就会被打开，所以它会查找一个 `EditText` 并且替换成一个新的 `code` 。
- 它会点击返回按钮。它会把新的值保存在preferences中，然后关闭Activity。
- 因为 `MainActivity` 的 `onResume` 会调用，请求会再调用一次。这时它会获取到新城市的forecast。
- 最后一行将会检测Toolbar我们看到的title是否是新的城市的title。

这不是一个toolbar的title的默认匹配器，但是 `Espresso` 是很容易扩展的，所以我们可以创建一个新的matcher来实现检测：

```
private fun withToolbarTitle(textMatcher: Matcher<CharSequence>)
: Matcher<Any> =
        object : BoundedMatcher<Any, Toolbar>(Toolbar::class.java
) {

    override fun matchesSafely(toolbar: Toolbar): Boolean {
        return textMatcher.matches(toolbar.title)
    }

    override fun describeTo(description: Description) {
        description.appendText("with toolbar title: ")
        textMatcher.describeTo(description)
    }
}
```

`matchSafely` 函数是我们检测的地方，而 `describeTo` 函数为matcher增加了一些新的信息。

这章特别有趣，因为我们看到了在Kotlin中怎么样去完美和谐地整合测试，它们可以没有任何问题地整合测试。查看代码然后你自己运行一下吧。

其它的概念

通过本书，我们讲解了Koltin语言中大部分的概念。但是其中某一些我们在这个App中没有使用到，我会把它们放到这章中来。这章中，我们回顾一些无关的内容，以便你在你自己的Kotlin项目中可以使用到它们。

内部类

在Java中，我们可以在类的里面再定义类。如果它是一个通常的类，它不能去访问外部类的成员（就如Java中的static）：

```
class Outer {
  private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

如果需要去访问外部类的成员，我们需要用 `inner` 声明这个类：

```
class Outer {
    private val bar: Int = 1
    inner class Inner{
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

枚举

Kotlin也提供了枚举（ `enums` ）的实现：

```
enum class Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

枚举可以带有参数：

```
enum class Icon(val res: Int) {
    UP(R.drawable.ic_up),
    SEARCH(R.drawable.ic_search),
    CAST(R.drawable.ic_cast)
}

val searchIconRes = Icon.SEARCH.res
```

枚举可以通过 `String` 匹配名字来获取，我们也可以获取包含所有枚举的 `Array` ，所以我们可以遍历它。

```
val search: Icon = Icon.valueOf("SEARCH")
val iconList: Array<Icon> = Icon.values()
```

而且每一个枚举都有一些函数来获取它的名字、声明的位置：

```
val searchName: String = Icon.SEARCH.name()
val searchPosition: Int = Icon.SEARCH.ordinal()
```

枚举根据它的顺序实现了 `Comparable` 接口，所以可以很方便地把它们进行排序。

密封（Sealed）类

密封类用来限制类的继承关系，这意味着密封类的子类数量是固定的。看起来就像是枚举那样，当你想在一个密封类的子类中寻找一个指定的类的时候，你可以事先知道所有的子类。不同之处在于枚举的实例是唯一的，而密封类可以有很多实例，它们可以有不同的状态。

我们可以实现，比如类似Scala中的 `Option` 类：这种类型可以防止null的使用，当对象包含一个值时返回 `Some` 类，当对象为空时则返回 `None` ：

```
sealed class Option<out T> {
    class Some<out T> : Option<T>()
    object None : Option<Nothing>()
}
```

有一件关于密封类很不错的事情是当我们使用 `when` 表达式时，我们可以匹配所有选项而不使用 `else` 分支：

```
val result = when (option) {
    is Option.Some<*> -> "Contains a value"
    is Option.None -> "Empty"
}
```

异常（Exceptions）

在Kotlin中，所有的 `Exception` 都是实现了 `Throwable` ，含有一个 `message` 且未经检查。这表示我们不会强迫我们在任何地方使用 `try/catch` 。这与Java中不太一样，比如在抛出 `IOException` 的方法，我们需要使用 `try-catch` 包围代码块。通过检查exception来处理显示并不是一个好的方法。像Bruce Eckel、Rod Waldhoff或Anders Hejlsberg等人可以给你关于这个更好的观点。

抛出异常的方式与Java很类似：

```
throw MyException("Exception message")
```

`try` 表达式也是相同的：

```
try{
    // 一些代码
}
catch (e: SomeException) {
    // 处理
}
finally {
    // 可选的finally块
}
```

在Kotlin中， `throw` 和 `try` 都是表达式，这意味着它们可以被赋值给一个变量。这个在处理一些边界问题的时候确实非常有用：

```
val s = when(x){
    is Int -> "Int instance"
    is String -> "String instance"
    else -> throw UnsupportedOperationException("Not valid type")
)
}
```

或者

```
val s = try { x as String } catch(e: ClassCastException) { null
}
```

结尾

感谢你阅读本书。通过本书，我们通过来实现一个Android App的例子来学习Kotlin。这个天气预报的App是一个不错的例子，它实现了大部分App需要的一些基本特性：一个主/从UI，通过API通信，数据库存储，shared preferences……

用这个方式不错的地方是你使用它们的使用学习到了大部分的Kotlin中重要的概念。我觉得新的语言在真正实践的时候更加容易被掌握。这是我主要的目标，参考书的确是一个解决一些标准问题的很好的工具，但是我们从头到尾阅读起来是很困难的。而且作为一些例子也是脱离于一个大的上下文环境，很难理解这些特性可以解决哪类问题。

而且实际上本书的其它的目标：展示给你看在Android中你会遇到的实际问题，并且使用Kotlin怎么去解决它们。一些Android开发者在处理异步、数据库或者处理Activity中非常冗长的listener时发现了很多的问题。通过作为一个例子的真正的App，我们遇到了很多问题并且学习到了新的语言和库的特性。

我希望这些目标已经达到了，并且我真的希望你不仅仅是在学习Kotlin，而且是在本书的阅读中得到享受。我被说服了，Kotlin对于Android开发者而言是目前最好的Java的替代者，我们会在接下来的时间中看到它的进步。当它事情发生时你将会是第一个上船的人，而且在你的圈子中你将会处于一个完美的参考人的位置。

本书已经结束了，但是这不是意味着它就死亡了。我将会一直根据最新版本保持更新（至少到1.0），根据你的留言和建议来检查并优化它。有什么想法可以在任何时候联系我，告诉我你的想法、你发现的错误、不够清晰的概念或者任何你顾虑的东西。

这几个月再写这本书的过程中经历了一个不可思议的旅行。我也学习到了很多，所以感谢你们的帮助让 `Kotlin for Android Developers` 这本书成为现实。

给你最好的祝福，

Antonio Leiva

- 网站：antonioleiva
- 邮箱：contact@antonioleiva.com
- Twitter：@lime_cl
- Google+：+AntonioLeivaGordillo